TURING 图灵程序设计丛书

[PACKT] PUBLISHING

最佳Hadoop入门图书，进入大数据处理的绝好途径，不容错过！

Hadoop基础教程

Hadoop Beginner's Guide

[英] Garry Turkington 著
张治起 译



人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

作者简介

Garry Turkington

拥有14年行业经验，其大部分时间都专注于大型分布式系统的设计与实现。目前，他在Improve Digital公司担任数据工程部副总裁和公司的首席架构师。他主要负责实现可以存储、处理并从公司海量数据中挖掘潜在价值的系统。在加入Improve Digital公司之前，他曾在Amazon英国公司领导几个软件开发团队，他们开发的系统用于处理Amazon为全世界所有对象创建的目录数据。在此之前，他还曾在英国和美国政府机关任职十年。

他在北爱尔兰的贝尔法斯特女王大学获得了计算机学士和博士学位，并在美国斯蒂文斯理工学院获得系统工程的工程硕士学位。

译者简介

张治起

Hadoop技术爱好者和研究者，对Hadoop技术有非常深刻的认识和理解，热切关注Hadoop和相关大数据处理技术。有着丰富的实践经验，热衷于技术分享，致力于不断探索揭开Hadoop的神秘面纱，帮助更多初学者接触和理解Hadoop。

TURING 图灵程序设计丛书



最佳Hadoop入门图书，进入大数据处理的绝好途径，不容错过！

Hadoop基础教程

Hadoop Beginner's Guide

[英] Garry Turkington 著
张治起 译

人民邮电出版社
北京

图书在版编目（CIP）数据

Hadoop基础教程 / (英) 特金顿 (Turkington,G.)
著 ; 张治起译. -- 北京 : 人民邮电出版社, 2014.1
(图灵程序设计丛书)
书名原文: Hadoop beginner's guide
ISBN 978-7-115-34133-4

Ⅰ. ①H… Ⅱ. ①特… ②张… Ⅲ. ①数据处理软件－
教材 Ⅳ. ①TP274

中国版本图书馆CIP数据核字(2013)第302148号

内 容 提 要

本书包括三个主要部分：第 1~5 章讲述了 Hadoop 的核心机制及 Hadoop 的工作模式；第 6~7 章涵盖了 Hadoop 更多可操作的内容；第 8~11 章介绍了 Hadoop 与其他产品和技术的组合使用。本书目的在于帮助读者了解什么是 Hadoop，Hadoop 是如何工作的，以及如何使用 Hadoop 从数据中提取有价值的信息，并用它解决大数据问题。

本书适用于有软件开发经验的技术人员。

◆ 著　　[英] Garry Turkington
译　　张治起
责任编辑　李　瑛
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　印刷

◆ 开本：800×1000　1/16
印张：20.25
字数：478千字　　2014年 1 月第 1 版
印数：1－3 500册　　2014年 1 月北京第 1 次印刷

著作权合同登记号　图字：01-2013-4680号

定价：65.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

版 权 声 明

前　　言

本书目的在于帮助读者理解Hadoop，并用它解决大数据问题。能使用Hadoop这样的数据处理技术进行工作是令人激动的。对大规模数据集进行复杂分析的能力，曾一度被大公司和政府机构所垄断，而现在通过免费的OSS（open source software，开源软件）即可获得这种能力。

该领域看上去有些复杂，并且变化节奏很快，所以理解这方面的基本知识让人望而生畏。这就是本书诞生的意义所在，它帮助读者了解什么是Hadoop，Hadoop是如何工作的，以及如何使用Hadoop从数据中提取有价值的信息。

除了解释Hadoop的核心机制，本书也会用几章内容来学习其他相关技术，这些技术要么用到了Hadoop，要么需要与Hadoop配套使用。我们的目标是，让读者不仅理解Hadoop是什么，还要理解如何在更宽泛的技术设施中使用Hadoop。

本书中提到的另一种技术是云计算的应用，尤其是AWS（Amazon Web Services，亚马逊网络服务）产品。本书中，我们将展示如何使用这些服务来承载Hadoop工作负载。这就意味着，读者无需购买任何物理硬件，就能处理大规模数据。

本书内容

本书包括3个主要部分：第1~5章讲述了Hadoop的核心机制及Hadoop的工作模式；第6~7章涵盖了Hadoop更多可操作的内容；第8~11章介绍了Hadoop与其他产品和技术的组合使用。每章内容具体如下。

第1章“绪论”。简要介绍了产生Hadoop和云计算的背景。如今看来，这些技术是如此重要。

第2章“安装并运行Hadoop”。指导读者完成本地Hadoop集群的安装，并运行一些示例作业。为了进行对比，在托管于亚马逊服务的Hadoop上执行同样的任务。

第3章“理解MapReduce”。深入研究Hadoop运行原理，揭示了MapReduce作业的执行方式，并展示了如何使用Java API编写MapReduce应用程序。

第4章“开发MapReduce程序”。通过对一个中等规模数据集案例的学习研究，演示如何着手处理和分析新数据源。

第5章“高级MapReduce技术”。介绍一些更复杂的应用MapRedece解决问题的方法，Hadoop似乎并不直接适用于这些问题。

第6章“故障处理”。详细检查Hadoop备受赞誉的高可用性和容错能力，通过强制结束进程和故意使用错误数据等方式故意制造破坏，以检验Hadoop在上述情况下的表现。

第7章“系统运行与维护”。从更具操作性的角度讲解Hadoop，这对于Hadoop集群的管理人员非常有用。本章在介绍一些最佳做法的同时，也描述了如何预防最糟糕的操作性灾难，因此系统管理员可以高枕无忧了。

第8章“Hive：数据的关系视图”。介绍Apache Hive，它允许使用类似SQL的语法对Hadoop数据进行查询。

第9章“与关系数据库协同工作”。探讨Hadoop如何与现有数据库融合，特别是如何将数据从一个数据库移到另一个数据库。

第10章“使用Flume收集数据”。介绍如何使用Apache Flume从多个数据源收集数据，并传送至Hadoop这样的目的地。

第11章“展望未来”。以更广泛的Hadoop生态系统概述结束全书，重点突出其他产品和技术的潜在价值。此外，还提供了一些如何参与Hadoop社区并获得帮助的方法。

准备工作

本书每章内容分别介绍该章用到的Hadoop相关软件包。但是，无论哪章内容都要用到运行Hadoop集群的硬件设施。

在最简单的情况下，一台基于Linux的主机可作为运行几乎全书所有示例的平台。任何可运行Linux命令行的先进操作系统都可满足需求，文中假设读者使用的是Ubuntu的最新发布版。

后面几章中的一些例子确实需要在多台机器上运行，所以读者需要拥有至少4台主机的访问权限。虚拟机也是可以的。虽然对于产品来讲，它们并非理想选择，但完全能够满足学习和研究的需要。

本书中，我们还将研究AWS，读者可以在EC2实例上运行所有示例。本书中，我们将更多地着眼于AWS针对Hadoop的具体用途。任何人都可以使用AWS服务，但需要一张信用卡进行注册！

目标读者

我们认为，本书读者想要学习更多关于Hadoop的实践知识。本书的主要受众是，有软件开发

经验却没有接触过Hadoop或类似大数据技术的人员。

对于那些想知道如何编写MapReduce应用的开发者，假设他们能轻松地编写Java程序并熟悉Unix命令行接口。本书还包含几个Ruby程序示例，但这些通常只是说明语言的独立性，并不要求读者成为一位Ruby专家。

对于架构师和系统管理员而言，本书也具有重大价值。它解释了Hadoop的工作原理，Hadoop在更广阔的系统架构中所处的位置，以及如何管理Hadoop集群。这类受众对一些复杂技术可能缺乏直接兴趣，如第4章和第5章。

约定

本书中，有几个经常出现的标题。

为了明确说明如何完成一个程序或任务，本书使用下面的格式详细描述操作步骤。

实践环节：标题

(1) 操作1

(2) 操作2

(3) 操作3

通常，需要一些额外的解释帮助读者理解这些指令，因此紧随其后的是下面的原理分析。

原理分析

这部分内容对任务运行原理或刚完成的指令进行解释说明。

本书还包含一些其他的学习辅助标记，包括：

随堂测验：标题

这是一些简短的多选题，目的在于帮助读者测试对相关内容的理解是否正确。

一展身手：标题

这部分内容设置一些实际问题，并为读者提供一些验证所学内容的方法。

本书使用多种文字风格来区分不同种类的信息。下面是一些例子还有对其意义的解释。

代码块设置如下：你也许注意到，我们使用Unix命令`rm`而不是DOS `del`命令移除`Drush`目录。

```
# * Fine Tuning
#
key_buffer = 16M
key_buffer_size = 32M
max_allowed_packet = 16M
thread_stack = 512K
thread_cache_size = 8
max_connections = 300
```

如果代码块的特定部分需要特别关注，相应行或内容会加粗显示。

```
# * Fine Tuning
#
key_buffer = 16M
key_buffer_size = 32M
max_allowed_packet = 16M
thread_stack = 512K
thread_cache_size = 8
max_connections = 300
```

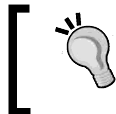
命令行的输入或输出会以如下形式显示。

```
cd /ProgramData/Propeople
rm -r Drush
git clone --branch master http://git.drupal.org/project/drush.git
```

新词或者重要的词会以黑体字显示。以菜单或对话框为例，读者在屏幕上看到的内容如下所示"在**Select Destination Location**（选择目的地址）页面，点击**Next**（下一步）按钮接受默认输出地址。"



警告或重要提示会出现在这样的方框里。



小窍门和技巧会以这样的形式出现。

读者反馈

我们非常欢迎来自读者的反馈。将你对本书的看法告诉我们，哪些内容是你喜欢的或哪些是你不喜欢的。读者反馈可以帮助我们不断提高书稿质量，这是非常重要的。

你只需向我们发送一封电子邮件，在邮件主题中提及书名即可反馈你的意见。我们的邮箱是：feedback@packtpub.com。

如果你擅长某个领域并对写作或者创作有兴趣，请参阅www.packtpub.com/authors上的作者指南。

客户支持

既然你购买了Packt的图书，我们将为你提供多种服务，使你本次购买物超所值。

下载示例代码

你可以通过http://www.packtpub.com网站的账号下载所购买Packt书籍的示例代码文件。或者登录网站http://www.packtpub.com/support进行注册，随后我们将通过电子邮件向你发送本书配套的示例代码文件。

勘误表

虽然作者尽最大努力保证本书内容的准确性，然而错误在所难免。如果你在Packt出版的书中发现了文字错误或是代码错误，衷心希望你能及时向我们反馈，并预致谢意。你的意见可以帮助我们改进图书的后续版本，并能帮助其他读者避免错误。请通过以下步骤提交你发现的错误之处：访问网站http://www.packtpub.com/submit-errata，选择相应图书，点击提交勘误表，登记你的勘误表详情。勘误表通过验证之后将上传到Packt网站，或添加到已有的勘误列表中。

关于盗版

互联网上的盗版问题由来已久，而且各种形式的媒介都存在这个问题。Packt出版社非常重视版权保护问题。如果你在互联网上看到了Packt图书的任何非法拷贝，无论它以何种形式存在，请立即向我们提供其位置或网站名称以便及时补救。

请通过邮箱copyright@packtpub.com与我们联系，并附上涉嫌盗版材料的链接。

你的帮助保障了Packt图书作者的权益，也让我们能持续提供高品质图书。

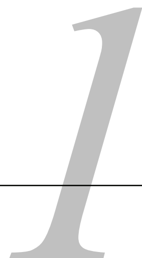
问题

如果你有关于本书任何方面的问题，可以通过questions@packtpub.com联系我们。我们将会尽力解决。

目 录

第 1 章

绪　　论



本书介绍一种大规模数据处理的开源框架——Hadoop。在深入探讨它的技术细节和应用之前，很有必要花点时间来了解Hadoop及其取得巨大成功的历史背景。

Hadoop并不是凭空想象出来的，它的出现源于人们创建和使用的数据量的爆炸性增长。在此背景下，不仅是庞大的跨国公司面临着海量数据处理的困难，小型创业公司同样如此。与此同时，其他历史变革改变了软件和系统的部署方式，除了传统的基础设施，人们开始使用甚至偏好于云资源。

本章将要探讨Hadoop出现的背景，并详细讲解Hadoop想要解决的问题和决定其最终设计的内在驱动因素。

本章包括以下内容：

- 概述大数据革命；
- 讲解Hadoop是什么以及如何从数据中获取有价值信息；
- 探秘云计算并了解AWS（Amazon Web Services，亚马逊网络服务）的功能；
- 了解大数据处理技术与云计算相结合带来的巨大威力；
- 概述本书其余章节内容。

现在我们开始吧！

1.1　大数据处理

审视现有技术，不难发现，所有技术都以数据为核心。作为用户，我们对富媒体的欲望与日俱增，比如观看的电影和创建并上传到网络的照片和视频。我们也常常在日常生活中，不经意地在网上留下一串数据。

不仅是数据总量在迅速增加，同时数据生成速率也在不断增加。从电子邮件到Facebook留言，从网上购物记录到网站链接，到处都是不断增长的大数据集。在此背景下，最大的挑战在于，如何从这些数据中提取最有价值的信息。有时是提取特定的数据元素，有时是分析数据间的关系或是判断一种趋势。

微妙的变化悄然发生，数据的使用方式变得越来越有意义。一段时间以来，大型公司已经意识到了数据的价值，并用它来提升服务质量。例如，Google在用户正在访问的网页上显示内容相关的广告，Amazon或Netflix向用户推荐合乎其口味和兴趣的新产品或新电影。

1.1.1　数据的价值

如果不会带来有价值的回报或者明显的竞争优势，这些大型企业是不会投资发展大数据处理技术的。应当从以下几个方面来认识大数据。

- 只有在数据集足够大的时候，某些问题才变得有意义。例如，在其他影响因素缺失的情况下，基于一个第三人的喜好推荐电影是不可能非常精准的。然而，当参考样本增加到100时，推荐成功的几率略有上升。而使用1000万人的观看记录，可以大幅提升获得推荐模型的可能性。
- 与之前的解决方案相比，大数据处理工具通常能够以较低的成本处理更大规模的数据。因此，能够在可接受的成本范围内执行以往花费极高的数据处理任务。
- 大规模数据处理的成本不仅体现在财务费用上，等待时间也是一个重要因素。如果一个系统能够处理所有到达数据，但是其平均处理时间以周为计量单位，那么该系统也是不可用的。大数据处理工具通过将增加的数据量分配到附加的硬件，来保证即使在数据量增加的情况下，处理时间也不会失控。
- 为了满足数据库中最大数据的需要，可能需要重新审视之前关于数据库形式或者其中数据结构的假设。
- 综合上述几点内容，足够大的数据集以及灵活的工具可以使之前无法想象的问题得到解答。

1.1.2　受众较少

前面讨论的从大数据中提取有价值信息用于改进服务质量的例子，往往属于大型搜索引擎和在线公司的创新模式。这是因为在早期发展过程中，大数据处理不仅成本高而且实现困难，超出了中小企业的能力范围。

同样，比大数据处理技术应用更为广泛的数据挖掘方法已经存在了很长一段时间，但是在大型企业和政府部门之外却从来没有真正得到推广使用。

这种情况的出现可能令人感到遗憾。但在过去，对于大多数小公司来讲却无关紧要，因为它们的数据量很少，并不需要投入大量的资金来处理这些数据。

然而，现如今数据量的增加不再局限于大型组织，许多中小型公司甚至一些个人收集到的数据也越来越多。他们也意识到这些数据中可能包含着正待发掘的价值。

在理解如何实现这一目标之前，很有必要了解奠定Hadoop系统基础的背景情况。

1. 经典的数据处理系统

造成大数据挖掘系统稀有且昂贵的根本原因是，将现有小型计算系统扩展为大数据处理系统是非常困难的。正如我们所见，一直以来，数据处理系统的处理能力一直受限于单台计算机的极限运算能力。

随着数据规模的增长，出现了两种常用的扩展系统的方法，通常称之为“向上扩展”和“向外扩展”。

- 向上扩展

在大多数企业，数据处理任务通常由相当昂贵的大型机来执行。随着数据规模的增长，向上扩展的方法就是将数据处理任务迁移到更大的服务器或者存储矩阵。即便以今天的视角来看，这种架构确实有效，但其硬件成本会很容易达到几十万甚至几百万美元。本章的后续内容对此有所涉及。

简单的向上扩展系统的优点是，系统的架构并不会随着数据量的增大而发生显著变化。尽管采用了更大型的部件，但部件之间的基本关系（例如数据服务器和存储矩阵）却依然保持一致。对于像商业数据库引擎这样的应用而言，可以通过软件来处理使用硬件带来的复杂性。但是从理论上来讲，数据处理能力是通过把相同的软件迁移到规模越来越大的服务器上来实现的。需要注意的是，把软件迁移到越来越多的处理器上也存在一定困难。同时，单台计算机的处理能力也受到现实条件的约束。因此，向上扩展的系统达到其极限后，无法进一步扩展数据处理的规模。

单一架构的数据处理器的规模不可能无限扩大。从概念上来讲，通过向上扩展系统的方式设计一个处理能力为1 TB、100 TB到1 PB数据的系统，可能仅需使用更大规模的相同部件。但随着数据规模的增长，这些定制硬件之间连接的复杂性可能与之前的廉价部件有所不同。

- 早期向外扩展的方法

向外扩展的方法并不通过升级系统的硬件来获得更强的处理能力，而是将数据处理任务分发给越来越多的机器。如果数据集的规模翻倍了，那就使用两台机器来处理，而不是一台有着两倍处理能力的机器。如果数据规模继续翻倍，那就使用四台机器来处理。

与向上扩展的方法相比，向外扩展系统的明显优势在于采购成本大大低于前者。大型机的硬

件成本随着处理能力的增长而激增——假设一台主机的采购成本为5000美元，那么一台具有10倍处理能力的机器可能要花掉百倍的钱。向外扩展系统方法的不足之处在于，需要确定一种策略来把数据处理任务分发给不同的机器，而经验证明具有上述用途的工具异常复杂。

因此，部署一套向外扩展的解决方案是一项浩大的工程。系统研发人员不仅要设计数据切分和重组机制，还要设计在处理器集群间调度工作和处理个别机器故障的逻辑。

2. 制约因素

除大型企业、政府和学术研究机构外，上述向上扩展和向外扩展的方法并没有得到广泛应用。通常，系统的采购成本很高，研发和维护这些系统的成本同样很高。这些因素导致这些系统很难被小型企业所接纳。此外，这些方法本身的缺陷也随着时间推移逐步显现。

- 随着向外扩展的系统规模变大或者向上扩展的系统所需CPU数量增多，由系统并发带来的系统复杂性问题日益明显。如何有效利用多台主机或多个CPU是一个大难题。要想在整个数据处理任务执行期间保持系统高效运作，需要付出极大的努力。
- 通常硬件性能（用摩尔定律描述）的提升在不同硬件上的表现大相径庭。CPU性能的提升速度远远超过了网络或硬盘性能提升的速度。CPU周期一度是系统中最有价值的资源，现在却并非如此。现代CPU的运行速度是20年前的数百万倍，然而内存和硬盘的速度却只提升了上千倍甚至上百倍。用这种高性能CPU很容易搭建一个现代系统，但在该系统中，存储系统提供数据的速率却无法满足CPU的工作需要。

1.1.3　一种不同的方法

上述场景中，有一些技术成功地解决了令人头疼的将数据处理系统扩展为大数据处理系统的问题。

1. 条条大路通罗马：向外扩展

如上所述，采用向上扩展的方法并不是一种开放式的策略。它受限于从主流的硬件供应商购买的单个服务器的规模，甚至专业的供应商都无法提供足够大的服务器。在某些情况下，工作负载的增量会超出单台整体向上扩展的服务器的能力，这时该怎么办呢？很遗憾，最好的办法就是使用两台大型服务器，而不是一台服务器。以此类推，就会有三台、四台，甚至更多服务器。或者换言之，在极端情况下，向上扩展架构的必然趋势是加入向外扩展的策略，将二者结合起来。尽管这样同时吸收了两种方法的部分优点，但也综合了两种方法的缺陷和成本：单一的方法要么需要昂贵的硬件，要么需要手工开发跨集群逻辑，而在混合架构中缺一不可。

向上扩展架构的终极趋势和成本曲线导致其在大数据处理领域鲜有应用，而向外扩展的架构却成了事实上的标准。



假如待解决的问题涉及的数据具有很强的内部交叉引用，并需保证事务完整性，基于大型计算机的向上扩展的关系数据库仍不失为一个好的选择。

2. 不共享任何内容

父母会花相当长的时间去教孩子，这是一个很好的共享案例。但是，不应将共享原则延伸到数据处理系统，这一原则既适用于数据也适用于硬件。

从概念上看，向外扩展架构刻意凸显了主机的独立性，每台主机处理整个数据集的一个子集，并输出最终结果的一部分。现实情况通常并非如此简单。相反，主机之间可能需要通信，多台主机可能会用到相同的一些数据片段。主机间相互依赖给系统带来了两个负面影响：瓶颈问题和增加了故障风险。

如果系统的每次运算都要调用同一块数据或同一个服务器，那么多个相互竞争的客户端访问同一数据或同一主机很可能造成系统延迟。例如，由25台主机组成的系统中，所有的主机都必须访问其中一台主机，整个系统的性能就会受限于这台关键主机的处理能力。

更糟糕的是，如果这个“热点”服务器或存储系统的关键数据失效，那么整个工作将会瞬间瘫痪。早期的集群解决方案往往证明了这一风险——即便使用多台服务器共同处理工作任务，却往往使用同一个共享存储系统保存所有数据。

与共享资源不同，一个系统的各个组成部分应当尽可能保持独立。这种情况下，每个部件都能正常工作，而不用理会其余部件是否忙于处理复杂的工作，或是否已处于故障状态。

3. 故障预期

上述原则提倡尽可能保持独立，然而潜在的弊端是要消耗更多的硬件。如果组成系统的个别组件会经常定期或不定期地发生故障，这是可以实现的。



读者可能经常听到“五个九”这样的词，它指的是99.999%的正常运行时间或者可用性。虽然从绝对意义上来讲，这是最好的可用性，但也要认识到，由许多这种设备组成的系统的整体可靠性可以相差很大，这取决于该系统能否容忍单个组件故障。

假设一套系统需要5台可靠性达99%的服务器才能运行。那么，该系统的可用性为0.99*0.99*0.99*0.99*0.99，也就是可用性为95%。但是如果单个服务器的可靠性仅为95%的话，那么整个系统的可靠性就下降到只有76%。

相反，如果在任意时刻，只要保证5台服务器中的1台正常工作，系统即运行正常。那么，系

统的可用性就高达99.999%。系统的正常运行时间与各个组件的可靠性临界点相关，这个观点有助于更直观地认识什么是系统可用性。



> 如果“系统可用性为99%”这样的表述有点抽象的话，可以从系统在给定时间段内的故障时间来考虑这一问题。举个例子来说明，99%的可用性相当于一年中有3.5天的故障时间，或者是一个月中有7个小时的故障时间。99%的可用性还是听上去那么美好吗?

接受故障的做法，往往是新手在充分理解大数据处理系统过程中感到最为困惑的一个方面。这也是它与向上扩展系统架构的方法区别最大的地方。大型的向上扩展的服务器成本昂贵的一个主要原因是，生产商要投入巨大努力来减少部件故障给系统带来的影响。即便是低端服务器也会有冗余电源，但在大型计算机机箱里，多个CPU安装在CPU卡上，这些CPU卡又通过多个背板连接至内存插槽和存储系统。大型计算机厂商经常通过各种形式的极端方法来展示其产品的健壮性，如将正在运行的系统部件直接拔出，甚至用枪向其射击。如果系统构建时，设计者考虑到了可能出现的故障并保证其发生的概率在可接受范围内，而不是将每次故障作为必须消除的危机，将会出现完全不同的体系结构。

4. 软件智能化，硬件傻瓜化

如果我们希望尽可能地灵活使用硬件集群，为许多并行工作流程提供托管服务，解决方案就是为软件注入智能并从硬件抽离智能。

在这种模式下，硬件被视为资源的集合，由软件层来负责向硬件分配特定的工作任务负载。这就使得硬件成为通用的，并且价格较为低廉，更易于获得。同时，有效利用硬件的功能转移到了软件，而关于如何有效执行任务的知识恰恰贮存在软件中。

5. 移动处理程序，而非移动数据

假设需要对一个很大的数据集（比如1000 TB，也就是1 PB）中的每块数据执行4项操作。下面看一下解决上述问题的系统的不同实现方法。

传统的基于大型计算机的向上扩展方案会用到一个巨大的服务器，它连接到一个同样令人印象深刻的存储系统。几乎可以肯定，该系统使用了诸如光纤信道之类的技术来最大限度地提高存储带宽。该系统可以完成上述任务，却会变成I/O密集型系统；即便是高端存储交换机也会限制从存储系统到主机的数据传输速率。

另一种方法基于之前提到的集群技术，该方法会用到一个由1000台机器组成的集群。这1000台主机被划分为4个象限，每个象限对应一种操作。每台主机都存有1 TB数据并负责执行四项操作之一。集群管理软件将会协调数据在集群间的流转，确保每块数据都经过4个处理过程。当每

块数据在其自身驻留的主机上进行一次运算之后，就会被传送到其他三个象限。因此，这个数据处理任务实际上消耗了3 PB的网络带宽。

请注意，CPU处理能力提升的速度已经超过了网络或硬盘技术。那么，难道这些是解决问题的最好办法吗？最近的经验表明，答案是否定的，另一种替代方法是移动处理过程而避免移动数据。使用刚才提到的集群技术，但不要分阶段处理数据，而是让1000个节点中的每个节点对其存储的数据执行全部4个处理阶段。如果幸运的话，此时只需将数据在硬盘上流转一次，而在网络上传输的将是二进制程序和状态报告，二者相对于有问题的实际数据集来讲，真是小巫见大巫。

如果一个有着1000个节点的集群听起来大得离谱，那么想像一下那些用于大数据解决方案的现代服务器的构成模块。每台这种单机都配有12个容量为1 TB甚至2 TB的硬盘。因为现代处理器是多核的，构建一个由50个节点组成的集群并非难事，这个集群有1 PB的存储能力，并有一个CPU专门用于处理从每个硬盘传出的数据。

6. 构建应用程序，而非基础架构

当思考上节中提到的问题时，多数人会将精力集中在数据迁移和数据处理的问题上。但是，曾经搭建过这种系统的人就会知道，大部分精力却是用在设计一些不太明显的因素的处理逻辑上，如任务调度、异常处理、协调配合等。

我们必须实现一些机制来确定在哪台主机上执行处理任务、实现处理过程、将子结果合并为整体结果，并且无法从以前的模型中获得多少帮助。我们需要明确地管理数据分区，这只是用一个难题换来了另一个难题。

上述问题与我们将强调的最新研究趋势相关：透明处理集群的大部分结构问题，让开发者专注于思考业务问题。基于明确定义的系统接口，开发者可以创建特定业务领域的应用程序；提供此类接口的框架将是开发者和系统效率的最佳组合。

1.1.4　Hadoop

得知上述方法都是Hadoop的主要特性之后，思维缜密的读者并不会感到惊讶。但直到现在，我还没有准确地回答什么是Hadoop。

1. 感谢Google

Hadoop起源于Google。Google公司于2003年和2004年发表了两篇描述Google技术的学术论文：谷歌文件系统（GFS）（http://research.google.com/archive/gfs.html）和MapReduce（http://research.google.com/archive/mapreduce.html）。它们提供了一个高效处理极大规模数据的平台。

2. 感谢Doug

与此同时，Doug Cutting正在研究开源的网页搜索引擎Nutch。他一直致力于系统原理的工作，当Google的GFS和MapReduce论文发表后，引起了他的强烈共鸣。Doug开始着手实现这些Google系统，不久之后，Hadoop诞生了。Hadoop早期以Lucene子项目的形式出现，不久之后成了Apache开源基金会的顶级项目。因此，从本质上来讲，Hadoop是一个实现了MapReduce和GFS技术的开源平台，它可以在由低成本硬件组成的集群上处理极大规模的数据集。

3. 感谢Yahoo

2006年，Yahoo雇用了Doug Cutting并迅速成为Hadoop项目最重要的支持者之一。除了经常推广一些全球最大规模的Hadoop部署之外，Yahoo允许Doug和其他工程师们在受雇于Yahoo期间，致力于Hadoop的工作。同时，Yahoo也贡献了一些公司内部开发的Hadoop改进和扩展程序。尽管Doug目前已经转到Cloudera（另一家大力支持Hadoop团体的创业公司）任职，Yahoo的Hadoop团队已经分拆为名叫Hortonworks的创业公司，Yahoo依然是Hadoop的一个主要支持者。

4. Hadoop的组成部分

作为一个顶级项目，Hadoop项目包含许多组件子项目。本书中将讨论其中几个子项目，最主要的两个子项目分别为Hadoop分布式文件系统（HDFS）和MapReduce。这两个子项目是对Google特有的GFS和MapReduce的直接实现。本书将详细讨论HDFS及MapReduce，但现在最好将它们视为一对独立而又互补的技术。

HDFS是一个可以存储极大数据集的文件系统，它是通过向外扩展方式构建的主机集群。它有着独特的设计和性能特点，特别是，HDFS以时延为代价对吞吐量进行了优化，并且通过副本替换冗余达到了高可靠性。

MapReduce是一个数据处理范式，它规范了数据在两个处理阶段（被称为Map和Reduce）的输入和输出，并将其应用于任意规模的大数据集。MapReduce与HDFS紧密结合，确保在任何情况下，MapReduce任务直接在存储所需数据的HDFS节点上运行。

5. 公共构建模块

HDFS和MapReduce都体现了一些上一节描述的体系原则。特别是：

- 都是面向廉价服务器（也就是说，中低端）集群设计的；
- 都是通过增加更多的服务器来实现系统扩展（向外扩展）；
- 都包含了检测和处理故障的机制；
- 都透明地提供了许多服务，从而使用户可以更专注于手头的问题；
- 都采用了同样的架构，其中驻留在物理服务器上的软件集群控制着系统运行的各个方面。

6. HDFS

HDFS（Hadoop Distributed File System，Hadoop分布式文件系统）与之前所见过的大部分文件系统都不太一样。HDFS是一个不具备POSIX兼容性的文件系统，这基本上意味着它不能提供像普通文件系统一样的保障。它也是一个分布式文件系统，这意味着它在众多节点上扩大了存储；在某些年以前的技术中，缺乏这种高效的分布式文件系统是一个限制性因素。HDFS的主要特点如下所示。

- HDFS通常以最小64 MB的数据块存储文件，这比之前多数文件系统中的4 KB~32 KB分块大得多。
- HDFS在时延的基础上对吞吐量进行了优化，它能够高效处理对大文件的读请求流，但不擅长对众多小文件的定位请求。
- HDFS对普遍的“一次写入，多次读取”的工作负载进行了优化。
- 每个存储节点上运行着一个称为DataNode的进程，它管理着相应主机上的所有数据块。这些存储节点都由一个称为NameNode的主进程来协调，该进程运行于一台独立主机上。
- 与磁盘阵列中设置物理冗余来处理磁盘故障或类似策略不同，HDFS使用副本来处理故障。每个由文件组成的数据块存储在集群中的多个节点，HDFS的NameNode不断地监视各个DataNode发来的报告，以确保发生故障时，任意数据块的副本数量都大于用户配置的复制因子。否则，NameNode会在集群中调度新增一个副本。

7. MapReduce

虽然MapReduce技术相对较新，但它建立在数学和计算机科学的众多基础工作之上，尤其是适用于每个数据元素的数据操作的描述方法。实际上，map和reduce函数的概念直接来自于函数式编程语言。在函数式编程语言中，map和reduce函数被用于对输入数据列表进行操作。

另一个关键的基本概念是“分而治之”。这个概念的基本原则是，将单个问题分解成多个独立的子任务。如果多个子任务能够并行执行，这种方法更为强大。在理想情况下，一个需要运行1000分钟的任务可以通过分解成1000个并行的子任务，在1分钟内即可完成。

MapReduce是一个基于上述原理的处理范式，它实现了从源数据集到结果数据集的一系列转换。在最简单的情况下，作业的输入数据作为map函数的输入，所得到的临时数据作为reduce函数的输入。开发人员只需定义数据转换形式，Hadoop的MapReduce作业负责并行地对集群中的数据实施所需转换。虽然基本的想法可能并无太大新意，但Hadoop的一个主要优势在于它如何将这些想法变成一个精心设计的可用平台。

传统的关系型数据库适用于符合定义模式的结构化数据。与之不同，MapReduce和Hadoop在半结构化或非结构化数据上表现最好。与符合刚性模式的数据不同，MapReduce和Hadoop仅要求将数据以一系列键值对的形式提供给map函数。map函数的输出是其他键值对的集合，reduce

函数收集汇总最终的结果数据集。

Hadoop为map和reduce函数提供了一个标准规范（即接口），上述规范的具体实现通常被称为**mapper**和**reducer**。一个典型的MapReduce作业包括多个mapper和reducer，通常这些mapper和reducer并不是很简单。开发人员将精力集中于表达从数据源到结果数据集的转化关系上，而Hadoop框架会管理任务执行的各个方面：并行处理，协调配合，等等。

最后一点可能是Hadoop最重要的一个方面。Hadoop平台负责执行数据处理的各个方面。在用户定义了任务的关键指标后，其余事情都变成了系统的责任。更重要的是，从数据规模的角度来看，同一个MapReduce作业适用于存储在任意规模集群上的任意大小的数据集。如果要处理的是单台主机上的1 GB数据， Hadoop会相应地安排处理过程。即便要处理的是托管在超过1000台机器上的1 PB数据，Hadoop依然同样工作。它会确定如何最高效地利用所有主机进行工作。从用户的角度来看，实际数据和集群的规模是透明的，除了处理作业所花费的时间受影响外，它们不会改变用户与Hadoop的交互方式。

8. 组合使用效果更佳

我们有可能已经体会到了HDFS和MapReduce各自的价值，但当它们组合使用时威力更强大。作为一个大规模数据存储平台，HDFS并非必须与MapReduce配套使用。尽管MapReduce可以从HDFS之外的数据源读取数据，并且对这些数据执行的处理操作与HDFS是一致的，但截至目前，将HDFS和MapReduce组合使用是最为常见的情况。

当执行MapReduce作业时，Hadoop需要决定在哪台主机执行代码才能最高效地处理数据集。如果MapReduce集群中的所有主机从单个存储主机或存储阵列获取数据，在很大程度上，在哪台主机执行代码并不重要，因为存储系统是一个会引发竞争的共享系统。但如果使用HDFS作为存储系统，基于移动数据处理程序比迁移数据本身成本更低的原则，MapReduce可以在目标数据的存储节点上执行数据处理过程。

最常见的Hadoop部署模型是将HDFS和MapReduce集群部署在同一组服务器上。其中每台服务器不仅承载了待处理数据及管理这些数据的HDFS组件，同时也承载了调度和执行数据处理过程的MapReduce组件。当Hadoop接收到作业后，它尽可能对驻留在主机上的数据调度进行优化，达到网络流量最小化和性能最大化的目标。

回想一下以前的例子，如何对分布在1000台服务器上的1 PB数据执行4个步骤的处理任务。MapReduce模型（以一种稍微简化和理想化的方式）对HDFS中每台主机上的每块数据执行map函数定义的处理操作，随后在reduce函数中，复用集群以收集各个主机的结果并转化为最终的结果集。

Hadoop的部分挑战在于，如何将单个问题分解成map函数和reduce函数的最佳组合。前述方法只适用于4阶段处理链可独立应用于每个数据元素的情况。在后续章节将会看到，有时要使

用多个MapReduce作业，其中某个作业的输出将作为下一个作业的输入。

9. 公共架构

正如之前所提到的，HDFS和MapReduce软件集群有着下列共同特点。

- 采用了相同的体系结构，用专门的主节点对工作节点集群进行管理。
- 每种情景的主节点（HDFS的主节点是NameNode，MapReduce的主节点是JobTracker）监视集群的运行状况并处理故障，处理方式包括移走数据块或者重新调度失败的作业。
- 每台服务器上的进程（HDFS的进程是DataNode，MapReduce的进程是TaskTracker）负责在物理主机上执行任务，包括从NameNode或者JobTracker接收指令和向其报告健康状态和运行状况。

作为一个小的术语点，主机或服务器通常指的是承载各种Hadoop组件的物理硬件。节点指的是作为集群组成部分的软件部件。

10. 优势与劣势

与其他工具一样，弄清楚在什么情况下选择使用Hadoop解决面临的问题非常重要。本书大部分内容会在之前大数据处理技术综述的基础上强调Hadoop的优势，但在早期阶段，理解在什么情况下Hadoop并非最好的选择也同样重要。

Hadoop选用的体系架构使其成为现在这样灵活且可扩展的数据处理平台。但是，与选择大多数架构或设计类似，必须理解一些由此引发的后果。其中最主要的是，Hadoop是一个批量处理系统。当对一个大数据集执行作业时，Hadoop框架会不断进行数据转换直到生成最终结果。由于采用了大的集群，Hadoop会相对快速地生成结果，但事实上，结果生成的速度还不足以满足缺乏耐心的用户的需求。因此，单独的Hadoop不适用于低时延查询，例如网站、实时系统或者类似查询。

当使用Hadoop处理大数据集时，安排作业、确定每个节点上运行的任务及其他所有必需的内务管理活动需要的时间开销在整个作业执行时间中是微不足道的。但是，对于小数据集而言，上述执行开销意味着，即使是简单的MapReduce作业都可能花费至少10秒钟。



HBase是广义Hadoop家族的另一位成员，它是另一种谷歌技术的开源实现。它提供了一个基于Hadoop的非关系型数据库，使用了多种方法提供低延迟的查询服务。

但是，难道Google和Yahoo不是Hadoop最强大的支持者吗？在它们的网站中，难道响应时间不是最重要的吗？答案是肯定的，这种现象强调了一个重要方面，即如何将Hadoop与其他技术结

合起来发挥共同优势。在一篇论文里（http://research.google.com/archive/googlecluster.html），谷歌简述了他们如何实时地使用MapReduce：在网络爬虫获取到更新的网页数据后，MapReduce对庞大的数据集进行处理，并基于此生成网页索引，这些检索被一组MySQL服务器用于为终端用户的搜索请求提供服务。

1.2　基于 Amazon Web Services 的云计算

云计算是本书介绍的另一个技术领域。书中以Amazon Web Services作为云计算平台的例子，介绍它的一些使用实例。但首先，我们需要了解一些围绕云计算的炒作和流行语。

1.2.1　云太多了

云计算已经成为一个被滥用的术语，可以说，云计算概念的滥用使其面临变得毫无意义的风险。因此，在本书中使用这个术语的时候，务必清楚并谨慎表达我们的真实意思。云有两个主要的特点：既是一种新的可选架构，也是一种解决成本的不同方法。

1.2.2　第三种方法

我们已经讨论了扩展数据处理系统的两种方法：向上扩展和向外扩展。但是迄今为止，前述讨论想当然地认为无论哪一种方法的实现，都需要系统开发团队购买、拥有、安装并管理物理硬件。云计算提供了第三种办法：用户把应用程序放到云端，让供应商处理系统扩展的问题。

当然，这并不总是那么简单。但是对于许多云服务来讲，这种模式才是真正的革命。用户根据一些公开的指南或者接口开发软件，然后把它部署到云平台，并允许它在成本允许的前提下根据需要扩展服务。但是由于扩展系统通常涉及成本，所以这是一个引人关注的命题。

1.2.3　不同类型的成本

这种云计算方式同时也改变了系统硬件的付费方式。通过降低硬件基础设施成本，构建能承载数千甚至数百万客户的平台，云服务提供商借此获得规模经济，同时所有用户都从中受益。作为使用者，不仅有人替你操心难度较大的工程问题，如扩展系统的问题，而且你可以根据需要的负载付费，而不必基于可能需要的最大工作负载来确定系统规模。相反，用户获得了弹性系统带来的好处，可以根据工作负载的需要使用或多或少的资源。

下面这个例子可以说明这一点。因为要计算税费和工资数据，许多公司的财务组在月底的工作量较大，而且通常年底的数据处理任务会更大。如果需要设计一个这种系统，购买多少硬件最为合适？如果购买的硬件仅够处理日常工作量，系统可能会在月底陷入困境，真正的麻烦可能出

现于年底处理任务运转起来的时候。如果以月底的工作量来度量系统规模，系统的部分硬件将会在一年的大部分时间处于闲置状态，而且仍然可能在处理年底任务的时候陷入困境。如果根据年底的工作量来确定系统规模，在一年的其余时间里，系统会有大量的闲置产能。除了考虑购买硬件的成本，还要考虑托管和运行成本（服务器的用电量可能占到其生命周期中成本的一大部分），你基本上浪费了大量的金钱。

云计算按需服务的特性可让用户的应用程序在一个规模不大的硬件上起步，然后随着时间的发展，向上或向下动态调整其硬件规模。基于按需付费的模式，用户成本随着其硬件使用情况而变化。用户拥有处理负载的能力，却无需购买足够处理峰值负载的硬件。

这种模型更微妙的特性在于，它大大降低了企业发布在线服务的进入成本。众所周知，一个无法满足需求并遭遇性能问题的新的热点服务，将很难恢复其发展势头和重新引起用户兴趣。例如，在2000年，一个企业要想成功地推出新品，需要在发布当天落实到位足够的容量来应对用户流量的激增，这既是他们所期望的，同时也是他们不愿看到的。将物理单元的成本考虑在内，在产品发布上很容易就会花掉数百万元。

如今，基于云计算，最初的基础设施成本甚至可以低至每月几十或几百美元，而且这只在需要流量的时候才会有所增加。

1.2.4　AWS：Amazon的弹性架构

Amazon Web Services（AWS）是Amazon 提供的一整套云计算服务。本书将会用到其中几种服务。

1. 弹性计算云（EC2）

Amazon的弹性计算云（EC2，参见http://aws.amazon.com/ec2/），其本质上是一个弹性服务器。在注册AWS和EC2之后，只需凭个人信用卡的有关信息即可获得专用虚拟机的访问权限，在这些服务器上可以很容易地运行多种操作系统，包括Windows和Linux的许多变种。

需要更多的服务器吗？只需启动更多的机器。需要更强大的服务器吗？只需切换到提供更高规格的服务器上，当然使用成本也随之上升。不仅如此，EC2提供了一系列免费服务，包括负载均衡器、静态IP地址、额外的高性能虚拟磁盘驱动器，和其他更多服务。

2. 简单存储服务（S3）

Amazon的S3（Simple Storage Service，简单存储服务，参见http://aws.amazon.com/s3/）是一种提供简单键值存储模式的存储服务。利用网络、命令行或者编程接口创建对象，这些对象可以是从文本文件到图片到MP3的一切对象。基于层次模型，用户能够在S3中存储并获取数据。在这个模型中，用户可以创建存放对象的桶（bucket）。每个桶都有一个唯一标识符，并且桶中的每个

对象都是唯一命名的。这种简单的策略成就了一个非常强大的服务，Amazon 负责数据的可靠性和可用性，以及服务扩展等各个方面。

3. 弹性 MapReduce（EMR）

Amazon的弹性MapReduce（EMR，参见 http://aws.amazon.com/elasticmapreduce/），从根本上来讲是以EC2和S3为基础的云端Hadoop。同样地，使用多种接口（CLI，Web控制台或API），用户可定义一些Hadoop工作流属性，例如所需的Hadoop主机数和源数据的位置等。EMR提供了MapReduce作业的Hadoop实现代码，之后虚拟的开始按钮被按下。

在其最令人印象深刻的模式中，EMR可以从S3获取源数据，利用基于EC2创建的Hadoop集群来处理这些数据，将结果返回到S3，然后终止Hadoop集群及承载它的EC2虚拟机的运行。当然，每个服务都是有成本的，通常以存储的数据量（以GB为单位）和服务器使用时间为计费基准，但无需专用硬件即可获得如此强大的数据处理能力，其功能真的是强大。

1.2.5　本书内容

本书中，我们将学习如何编写MapReduce程序做一些重要的数据转换，以及如何在本地管理和AWS托管的Hadoop集群上运行MapReduce程序。

我们不仅会把Hadoop视为执行MapReduce作业的引擎，同时也会探索将Hadoop融入企业其他基础设施和系统的方法。我们会看到一些共同的结合点，如从Hadoop和关系型数据库中获取数据，以及如何使Hadoop看起来更像是一个关系数据库。

双管齐下

本书不会将讨论局限在EMR或Amazon EC2托管的Hadoop。我们在讨论如何创建和管理本地Hadoop集群（在Ubuntu Linux系统上）的同时，也会演示如何通过EMR将处理过程推入云端。

这样做的原因来自两方面：首先，虽然EMR使Hadoop更易访问，但只有当手工管理集群时，才能充分体会Hadoop技术的各项特点。尽管在手工模式下也可以使用EMR，然而通常使用本地集群来进行学习研究。其次，使用托管Hadoop还是本地Hadoop并不是一个非此即彼的选择，有时用户会担忧对一个独立的外部供应商的过度依赖，所以许多企业混合使用本地和云托管的Hadoop服务。从实践来讲，在本地进行开发和小规模测试更为方便，然后再根据产品规模将其部署到云端。

在后面的一些章节中，我们会讨论与Hadoop组合使用的其他产品。在这部分内容中，本书只给出一些本地集群的例子，因为无论Hadoop部署在何处，工作原理都是不变的。

1.3　小结

本章介绍了关于大数据、Hadoop和云计算的大量知识。

专门探讨了大数据的出现以及它对数据处理方法及系统架构的改变，这一变革几乎可以让任何组织实现曾经昂贵得令人望而却步的技术。

本章还回顾了，作为一个灵活而又功能强大的海量数据处理平台，Hadoop的产生历史和构建方式。我们还研究了云计算提供的另一种系统架构方式。这种方式从前期巨额成本和直接的物理责任转变为按需付费模式，并依赖云服务提供商来提供硬件、管理服务和扩展系统。我们也明白了什么是Amazon Web Services，以及弹性MapReduce服务怎样利用其他AWS服务实现云端Hadoop。

我们还讨论了本书的目标，以及如何学习本地Hadoop和AWS托管的Hadoop。

现在，我们已经掌握了Hadoop的基础知识，并了解了这项技术的起源和优势。我们需要动手让系统运行起来，这也是第2章将要做的：安装并运行Hadoop。

第2章

安装并运行Hadoop



既然我们已经研究了大数据处理带来的机遇和挑战，并且了解了Hadoop的优势所在，现在就来安装并运行Hadoop吧。

本章包括以下内容：

- 学习如何在本地Ubuntu主机安装并运行Hadoop；
- 运行一些Hadoop程序实例并熟悉Hadoop系统；
- 注册使用Amazon Web Services产品（如EMR）所需要的账号；
- 在Elastic MapReduce上创建符合需求的Hadoop集群；
- 研究本地Hadoop集群与托管Hadoop集群的关键区别。

2.1 基于本地 Ubuntu 主机的 Hadoop 系统

为了研究云外的Hadoop，本节将以一台或多台Ubuntu主机为例。一台主机（一台物理计算机或虚拟机）即可满足运行Hadoop所有组件和研究MapReduce的需要。然而，企业产品集群很可能由多台机器组成，所以如果能在部署于多台主机的Hadoop集群进行开发，读者将获得很好的经验。然而，对于起步而言，单台主机就足够了。

我们将讨论的内容并非仅限于Ubuntu操作系统，Hadoop能够在任何Linux操作系统上运行。很明显，如果读者使用了Ubuntu之外的操作系统，可能需要修改一下环境配置，但区别应该不大。

其他操作系统

Hadoop在其他平台上运行效果良好。Windows和Mac OS X都是开发者的热门选择。Windows只能作为Hadoop的开发平台，而Mac OS X没有得到Hadoop的正式支持。

如果选择使用这类平台，情况将与使用其他Linux发行版本相类似。在这两个平台上使用

Hadoop处理工作任务的方法是相同的,但需要使用操作系统自带的特定机制设置环境变量和类似任务。Hadoop FAQ包含一些其他平台的信息，如果读者打算在其他平台运行Hadoop，应当首先参考Hadoop FAQ（参见http://wiki.apache.org/hadoop/FAQ）解决相关问题。

2.2 实践环节：检查是否已安装 JDK

Hadoop是用Java实现的，所以需要首先在Ubuntu主机上安装最新的Java开发工具包（JDK）。执行下列步骤检查JDK是否可用：

(1) 首先，打开一个终端并输入以下内容来检查JDK是否可用：

```
$ javac
$ java -version
```

(2) 如果上述命令返回“不存在这样的文件或路径”或者类似的错误，或者第二个命令返回“打开JDK”的提示，很可能需要下载完整的JDK。JDK可以从Oracle下载页面获得，地址是http://www.oracle.com/technetwork/java/javase/downloads/index.html，从中选择最新发布的版本。

(3) 一旦安装了Java之后，将JDK/bin路径添加到用户路径，并用下列命令设置JAVA_HOME环境变量，注意将Java版本号修改成读者使用的Java版本。

```
$ export JAVA_HOME=/opt/jdk1.6.0_24
$ export PATH=$JAVA_HOME/bin:${PATH}
```

原理分析

上述步骤确保用户安装了正确的Java版本，并可以在命令行下调用，而无须使用冗长的路径名指明安装位置。

请注意，上面的命令只对当前正在运行的shell产生影响。这些设置将在注销用户，关闭shell，或重新启动系统之后清除。为了确保相同的设置始终有效，可以把这些设置添加到读者选中的shell启动文件内。例如，对于Bash shell来讲，其启动文件为`.bash _profile`，对于TCSH来讲，其启动文件为`.cshrc`。

我所青睐的另一种方法是，把所有必需的配置放入一个独立的文件，然后从命令行显式地调用此文件。例如：

```
$ source Hadoop_config.sh
```

这种方法允许读者在同一账户内保持多个安装文件，而不致于使shell启动过程过于复杂。更

不用说，某些应用程序所需的配置确实可能与其他程序的配置不兼容。别忘了在每次开启会话前载入该文件。

安装Hadoop

对于初学者来讲，Hadoop最令人困惑的一个方面在于它众多的组件、工程、子工程，以及它们之间的内在关系。事实上，随着时间的推移，这些组件都发生了变化，然而却并没有使这一切变得更容易理解。不过，从今往后，访问网站http://hadoop.apache.org会看到Hadoop包括曾提到过的3个主要工程：

- Common
- HDFS
- MapReduce

通过第1章的解释，我们应该对后两项内容较为熟悉。Common工程是Hadoop项目的核心部分，包括一组运行库和各种工具，它们帮助Hadoop实际运作。目前重要的是，标准的Hadoop发行版软件包含了这3个项目的最新版本，它们的组合正是运行Hadoop所需要的。

各版本的注意事项

从0.19到0.20版本，Hadoop发生了重大的变化。尤其是，用于开发MapReduce应用的API迁移到了一组新API。本书中，我们将主要使用新API，但是后面的章节中也包含了一些旧API的例子，这是因为现在并没有将现有的所有功能都移植到新API。

自从0.20分支被重命名为1.0，Hadoop的版本管理也变得复杂。0.22和0.23分支仍然存在，事实上还包含了一些1.0分支所不包含的特性。在写作本书的时候，由于1.1和2.0分支被用作将来的开发版本，所以事情变得更加清晰。由于大多数现有的系统和第三方工具是针对0.20分支开发的，本书中，我们将使用Hadoop 1.0作为例子。

2.3　实践环节：下载 Hadoop

执行以下操作来下载Hadoop。

(1) 访问Hadoop下载页面，网址为http://hadoop.apache.org/common/releases.html，获取1.0.x分支的最新稳定版本。本书写作时，最新稳定版为1.0.4。

(2) 选择一个本地镜像，之后以类似`hadoop-1.0.4-bin.tar.gz`的名字下载文件。

(3) 使用如下命令将文件拷贝到Hadoop的安装路径（例如，`/usr/local`）：

```
$ cp Hadoop-1.0.4.bin.tar.gz /usr/local
```

(4) 使用如下命令将文件解压缩：

```
$ tar -xf hadoop-1.0.4-bin.tar.gz
```

(5) 为Hadoop的安装路径添加方便使用的符号链接。

```
$ ln -s /usr/local/hadoop-1.0.4 /opt/hadoop
```

(6) 现在，需要将Hadoop的二进制目录添加到PATH变量，并设置HADOOP_HOME环境变量，就如设置Java一般。

```
$ export HADOOP_HOME=/usr/local/Hadoop
$ export PATH=$HADOOP_HOME/bin:$PATH
```

(7) 在Hadoop安装路径下，进入conf目录并编辑Hadoop-env.sh文件。搜索JAVA_HOME并取消该行的注释，修改其路径使其指向JDK的安装路径，如前所述。

原理分析

上述步骤确保Hadoop已经安装并可通过命令行调用。通过设置path及环境变量，我们可以使用Hadoop的命令行工具。Hadoop配置文件的修改，是安装过程所需的唯一改动，为的是与主机设置相结合。

如前所述，读者应该把export命令放入shell启动文件，或者启动会话时指定的独立配置脚本。

不要纠结于这里讲到的某些细节，后续内容还会介绍Hadoop的安装和使用。

2.4 实践环节：安装 SSH

执行下列步骤以安装SSH。

(1) 通过下列命令创建一对OpenSSL密钥对：

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/hadoop/.ssh/id_rsa):
Created directory '/home/hadoop/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/hadoop/.ssh/id_rsa.
Your public key has been saved in /home/hadoop/.ssh/id_rsa.pub.
...
```

(2) 使用下列命令将新生成的公钥复制至已授权秘钥列表：

```
$ cp .ssh/id _rsa.pub    .ssh/authorized_keys
```

(3) 连接本地主机：

```
$ ssh localhost
The authenticity of host 'localhost (127.0.0.1)' can't be established.
RSA key fingerprint is b6:0c:bd:57:32:b6:66:7c:33:7b:62:92:61:fd:ca:2a.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (RSA) to the list of known hosts.
```

(4) 确认无密码的SSH可以运行：

```
$ ssh localhost
$ ssh localhost
```

原理分析

由于Hadoop需要在一台或多台计算机上的多个进程之间通信，我们需要确保正在使用Hadoop的用户不输入密码即可连接到所需的每台主机。通过创建有一个空口令**Secure Shell**（SSH）的密钥对来实现这一点。我们使用ssh-keygen命令启动这一进程，并接受所提供的缺省设置。

一旦创建了密钥对，需要将新生成的公钥添加到可信密钥的存储列表。这就意味着，当试图连接这台机器时，公钥会被信任。然后，使用ssh命令连接本地机器，应该会获得一个如上述显示的关于信任主机证书的警告。确认后，我们应该能够连接而不再需要密码或出现提示。



> 请注意，稍后使用一个完全分布式模式的集群时，我们需要确保集群中每台主机的Hadoop用户账号具有相同的密钥设置。

配置并运行Hadoop

到目前为止，这都非常简单，只涉及下载和系统管理。现在，终于可以和Hadoop打交道了。我们将运行一个简单的例子，以表明Hadoop正在运行。接下来的步骤需要执行额外的配置和设置，但是也说明到目前为止，所有一切安装和设置都是正确的。

2.5 实践环节：使用 Hadoop 计算圆周率

现在，我们使用一个简单的Hadoop示例程序计算圆周率的值。眼下，这主要是为了验证安装，同时展示MapReduce作业如此之快的执行速度。假设HADOOP_HOME/bin目录已存在于用户的PATH变量，请键入以下命令：

```
$ Hadoop jar hadoop/hadoop-examples-1.0.4.jar    pi 4 1000
Number of Maps    = 4
Samples per Map = 1000
Wrote input for Map #0
Wrote input for Map #1
Wrote input for Map #2
Wrote input for Map #3
Starting Job
12/10/26 22:56:11 INFO jvm.JvmMetrics: Initializing JVM Metrics with processName=
JobTracker, sessionId=
12/10/26 22:56:11 INFO mapred.FileInputFormat: Total input paths to process : 4
12/10/26 22:56:12 INFO mapred.JobClient: Running job: job_
local_0001

12/10/26 22:56:12 INFO mapred.FileInputFormat: Total input paths to process : 4
12/10/26 22:56:12 INFO mapred.MapTask: numReduceTasks: 1

...
12/10/26 22:56:14 INFO mapred.JobClient:    map 100% reduce 100%
12/10/26 22:56:14 INFO mapred.JobClient: Job complete: job_
local_0001
12/10/26 22:56:14 INFO mapred.JobClient: Counters: 13
12/10/26 22:56:14 INFO mapred.JobClient:    FileSystemCounters
...
Job Finished in 2.904 seconds
Estimated value of Pi is 3.14000000000000000000
$
```

原理分析

上述例程的输出包含了大量信息，当在屏幕上获得完整的输出时甚至会有更多的信息。现在，让我们一步一步进行分析，无需为Hadoop的状态输出感到困惑，因为后续内容会对它进行专门介绍。首先要弄清楚的是一些术语：每个Hadoop程序就是一个作业，它会创建多个任务来完成自己的工作。

查看输出，我们把它宽泛地概括为3个部分：

- 启动作业
- 作业的执行状态
- 作业结果的输出

在上述例子中可以看到，这个作业创建了4个任务来计算圆周率，整个作业的结果将是这些子结果的组合。这种模式听起来应该很熟悉，与第1章中讲到的类似。这种模型将一个较大的作业拆分成较小的任务来执行，然后将结果整合起来。

随着作业的执行，出现了大部分输出，它们提供了显示作业进度的状态消息。工作成功完成后，作业将打印出一些计数器和其他数据。事实上，这个例子的不同寻常之处在于，很少见到

MapReduce作业的结果直接显示在控制台上。这并不是Hadoop的一个局限性，而是由于，处理大数据集的作业通常会产生相当多的输出数据，并不适合简单地在屏幕上显示。

恭喜你，成功地完成了第一个MapReduce作业！

3 种模式

我们渴望在Hadoop上运行作业，却回避了一个重要的问题：应该在何种模式下运行Hadoop？Hadoop有3种运行模式，各种模式下，Hadoop组件的运行场所有所不同。回想一下，HDFS包括一个NameNode，它充当着集群协调者的角色，是一个或多个用于存储数据的DataNode的管理者。对于MapReduce而言，JobTracker是集群的主节点，它负责协调多个TaskTracker进程执行的工作。Hadoop以如下3种模式部署上述组件。

- **本地独立模式**：像前面计算圆周率的例子一样，如果不进行任何配置的话，这是Hadoop的默认工作模式。在这种模式下，Hadoop的所有组件，如NameNode、DataNode、JobTracker和TaskTracker，都运行在同一个Java进程中。
- **伪分布式模式**：在这种模式下，Hadoop的各个组件都拥有一个单独的Java虚拟机，它们之间通过网络套接字通信。这种模式在一台主机上有效地产生了一个具有完整功能的微型集群。
- **完全分布式模式**：在这种模式下，Hadoop分布在多台主机上，其中一些是通用的工作机，其余的是组件的专用主机，比如NameNode和JobTracker。

每种模式都有其优点和缺点。完全分布式模式显然是唯一一种可以将Hadoop扩展到机器集群的方式，但它需要更多的配置工作，更不用提所需要的机器集群。本地或独立模式的设置工作是最简单的，但它与用户的交互方式不同于全分布式模式的交互方式。一般情况下，本书更喜欢使用伪分布式模式，即使程序只需在一台主机上运行。这是因为，Hadoop在伪分布式模式下执行的操作与其在更大的集群上的运作几乎是相同的。

2.6　实践环节：配置伪分布式模式

查看Hadoop安装包中的conf目录。那里有很多配置文件，但只需对其中3个文件进行修改：core-site.xml、hdfs-site.xml和mapred-site.xml.

(1) 如下所示，修改core-site.xml文件：

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
```

```
<property>
<name>fs.default.name</name>
<value>hdfs://localhost:9000</value>
</property>
</configuration>
```


(2) 如下所示，修改hdfs-site.xml文件：

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
<name>dfs.replication</name>
<value>1</value>
</property>
</configuration>
```

(3) 如下所示，修改mapred-site.xml文件：

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
<name>mapred.job.tracker</name>
<value>localhost:9001</value>
</property>
</configuration>
```

原理分析

首先，请注意这些配置文件的通用格式。显然，它们都是XML文件，单个配置元素包含有多个属性说明。

属性说明总是包含名称和值元素，可能会有可选注释，这并没有在前面的代码中展示。

在这里，我们需要设置3个配置变量。

- 变量dfs.default.name保存了NameNode的位置，HDFS和MapReduce组件都需要它。这就是它出现在core-site.xml文件中而不是hdfs-site.xml文件中的原因。
- 变量dfs.replication指定了每个HDFS数据块的复制次数。回想一下第1章中讲述的内容，HDFS确保每个数据块被复制到多台不同主机（通常是3台），以此方式处理故障。由于我们只有一台主机和一个伪分布式模式的DataNode，将此值修改为1。

- 如同`dfs.default.name`保存了NameNode的位置，变量`mapred.job.tracker`保存了JobTracker的位置。因为只有MapReduce组件需要知道这个位置，所以它出现在`mapred-site.xml`文件中。



当然，读者可以随意修改要使用的端口号，但9000和9001是Hadoop的常用默认端口。

NameNode和JobTracker的网络地址指定了实际的系统请求应当到达的端口。这些位置都不面向用户，所以不用担心您的Web浏览器会指向他们。不久，我们将会讲解Web界面。

配置根目录并格式化文件系统

假如我们选择了伪分布式或完全分布式模式，在启动Hadoop集群之前需要执行两个步骤。

(1) 设置存储Hadoop文件的根目录。

(2) 格式化HDFS文件系统。



准确地讲，我们不需要修改默认目录，但是稍后会讲到，最好尽早考虑这一点。

2.7　实践环节：修改 HDFS 的根目录

首先来设置HDFS的根目录，它指定了HDFS在本地文件系统保存其全部数据的位置。执行以下步骤。

(1) 创建Hadoop保存数据的目录：

```
$ mkdir /var/lib/hadoop
```

(2) 确保任何用户都可在此目录写入数据：

```
$ chmod 777 /var/lib/hadoop
```

(3) 再次修改core-site.xml文件，添加下列属性：

```
<property>
<name>hadoop.tmp.dir</name>
<value>/var/lib/hadoop</value>
</property>
```

原理分析

由于我们将在Hadoop中存储数据，同时所有组件都运行在本地主机，这些数据都需要存储在本地文件系统的某个地方。不管选择了何种模式，Hadoop默认使用`hadoop.tmp.dir`属性作为根目录，所有文件和数据都将写入该目录。

例如，MapReduce使用根目录下的`/mapred`目录，HDFS使用`/dfs`目录。危险的是，`hadoop.tmp.dir`的默认值为`/tmp`，一些版本的Linux系统会在每次重新启动时删除`/tmp`的内容。所以，明确规定数据留存的位置更为安全。

2.8　实践环节：格式化 NameNode

无论是在伪分布式模式还是完全分布式模式下，在首次启动Hadoop之前，都需要格式化Hadoop将要用到的HDFS文件系统。输入如下命令：

```
$    hadoop namenode -format
```

该命令的输出应如下所示：

```
$ hadoop namenode -format
12/10/26 22:45:25 INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG:    host = vm193/10.0.0.193
STARTUP_MSG:    args = [-format]
...
12/10/26 22:45:25 INFO namenode.FSNamesystem: fsOwner=hadoop,hadoop
12/10/26 22:45:25 INFO namenode.FSNamesystem: supergroup=supergroup
12/10/26 22:45:25 INFO namenode.FSNamesystem: isPermissionEnabled=true
12/10/26 22:45:25 INFO common.Storage: Image file of size 96 saved in 0 seconds.
12/10/26 22:45:25 INFO common.Storage: Storage directory /var/lib/hadoop-
hadoop/dfs/name has been successfully formatted.
12/10/26 22:45:26 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at vm193/10.0.0.193
$
```

原理分析

这并不是一个令人非常兴奋的输出，因为这个步骤只是使我们今后能够使用HDFS。然而，它有助于我们对HDFS的理解，它只是一个文件系统，没有什么神秘的。就像使用任何操作系统的任何新的存储设备之前，我们都需要对其进行格式化操作。HDFS同样如此，最初有一个默认的存放文件系统数据的位置，但没有等同于文件系统索引的实际数据。



每次都需要执行格式化！

假如你的Hadoop和我的类似，在重新安装Hadoop的时候，经常会犯一系列的简单错误。因为很容易忘记格式化NameNode，于是就会在第一次尝试使用Hadoop的时候收到一系列故障信息。

但只执行一次格式化！

格式化NameNode的命令可以执行多次，但是这样会使所有的现有文件系统数据受损。只有在Hadoop集群关闭和你想进行格式化的情况下，才能执行格式化。但在其他大多数情况下，格式化操作会快速、不可恢复地删除HDFS上的所有数据。它在大型集群上的执行时间更长。所以一定要小心！

启动并使用Hadoop

在完成所有安装及配置步骤之后，现在启动集群并用它实际做一些事情。

2.9　实践环节：启动 Hadoop

在本地模式中，所有Hadoop组件只在作业的生命周期运行；伪分布式或者全分布式模式的Hadoop集群则与此不同，其组件以长期运行的进程的形式存在。在使用HDFS或MapReduce之前，需要事先启动所需组件。键入以下命令，输出应该如下所示（其中命令以$为前缀）：

(1) 输入第一个命令：

```
$ start-dfs.sh
starting namenode, logging to /home/hadoop/hadoop/bin/../logs/
hadoop-hadoop-namenode-vm193.out
localhost: starting datanode, logging to /home/hadoop/hadoop/
bin/../logs/hadoop-hadoop-datanode-vm193.out
localhost: starting secondarynamenode, logging to /home/hadoop/
hadoop/bin/../logs/hadoop-hadoop-secondarynamenode-vm193.out
```

(2) 输入第二个命令：

```
$ jps
9550 DataNode
9687 Jps
9638 SecondaryNameNode
9471 NameNode
```

(3) 输入第三个命令：

```
$ hadoop dfs -ls /
Found 2 items
```

```
drwxr-xr-x    -hadoop supergroup    0 2012-10-26 23:03 /tmp
drwxr-xr-x    -hadoop supergroup    0 2012-10-26 23:06 /user
```

(4) 输入第四个命令：

```
$ start-mapred.sh
starting jobtracker, logging to /home/hadoop/hadoop/bin/../logs/hadoop-hadoop-
jobtracker-vm193.out
localhost: starting tasktracker, logging to /home/hadoop/hadoop/bin/../logs/
hadoop- hadoop-tasktracker-vm193.out
```

(5) 输入第五个命令：

```
$ jps
9550 DataNode
9877 TaskTracker
9638 SecondaryNameNode
9471 NameNode
9798 JobTracker
9913 Jps
```

原理分析

顾名思义，`start-dfs.sh`命令启动HDFS的必要组件。这些组件包括管理文件系统的NameNode和一个保存数据的DataNode。SecondaryNameNode是一个可用性辅助程序，我们将在后面的章节讨论它。

启动这些组件后，我们使用JDK的`jps`工具查看哪个Java进程正在运行。从输出来看，系统运转正常，我们随后使用Hadoop的`dfs`工具列出HDFS文件系统的根目录。

在此之后，我们使用`start-mapred.sh`启动MapReduce的组件，这次会启动JobTracker和一个TaskTracker，然后再次使用`jps`来验证结果。

在稍后阶段，我们还将使用一个组合的`start-all.sh`文件。但在初期，执行两阶段的启动是非常有用的，这样更容易检验集群的配置是否正确。

2.10 实践环节：使用 HDFS

如前面例子所示，HDFS提供了一套看似熟悉的接口，用户可以使用类似于Unix系统的命令，来操作文件系统上的文件和目录。键入以下命令试验一下。

输入下列命令：

```
$ hadoop -mkdir /user
$ hadoop -mkdir /user/hadoop
```

```
$ hadoop fs -ls /user
Found 1 items
drwxr-xr-x    - hadoop supergroup    0 2012-10-26 23:09 /user/Hadoop
$ echo "This is a test." >> test.txt
$ cat test.txt
This is a test.
$ hadoop dfs -copyFromLocal test.txt    .
$ hadoop dfs -ls
Found 1 items
-rw-r--r--    1 hadoop supergroup    16 2012-10-26 23:19/user/hadoop/
test.txt
$ hadoop dfs -cat test.txt
This is a test.
$ rm test.txt
$ hadoop dfs -cat test.txt
This is a test.
$ hadoop fs -copyToLocal test.txt
$ cat test.txt
This is a test.
```

原理分析

这个例子展示了Hadoop实用工具中的`fs`子命令的用法。请注意，`dfs`和`fs`命令是相同的。像大多数文件系统一样，Hadoop为每个用户保留一个主目录。这些主目录都位于HDFS上的`/user`路径下。在继续深入之前，假如主目录不存在的话，我们需要自己创建主目录。

然后，在本地文件系统创建一个简单的文本文件，并使用`copyFromLocal`命令将其复制到HDFS，并通过`-ls`和`-cat`实用程序来检查文件是否存在并查看其内容。在Unix系统中，未带参数的`-ls`命令指定是用户主目录，相对路径（不以`/`开头）指的也是那个位置，因此，不难看出，用户主目录的别名是`.`。

然后，从本地文件系统删除文件，使用`-copyToLocal`命令从HDFS将其复制回到本地文件系统，用本地`cat`工具检查其内容。



如上述例子所示，混合使用HDFS和本地文件系统命令十分强大，很容易在HDFS上执行原本为本地文件系统设计的命令，反之亦然。所以，要多加小心，尤其是在删除文件的时候。

还有一些HDFS的操作命令，试着用`hadoop fs -help`来获取详细列表。

2.11　实践环节：MapReduce 的经典入门程序——字数统计

随着时间的推移，许多应用程序都有一个经典例子，它是所有初学者指南都要用到的。对

Hadoop而言，这就是字数统计程序WordCount——一个Hadoop自带的例子，它用来统计一个输入文本文件中每个词的出现频率。

(1) 首先执行下列命令：

```
$ hadoop dfs -mkdir data
$ hadoop dfs -cp test.txt data
$ hadoop dfs -ls data
Found 1 items
-rw-r--r--    1 hadoop supergroup    16 2012-10-26 23:20 /
user/hadoop/data/test.txt
```

(2) 现在执行这些命令：

```
$ Hadoop Hadoop/hadoop-examples-1.0.4.jar    wordcount data out
12/10/26 23:22:49 INFO input.FileInputFormat: Total input paths to process : 1
12/10/26 23:22:50 INFO mapred.JobClient: Running job:
job_201210262315_0002
12/10/26 23:22:51 INFO mapred.JobClient:    map 0% reduce 0%
12/10/26 23:23:03 INFO mapred.JobClient:    map 100% reduce 0%
12/10/26 23:23:15 INFO mapred.JobClient:    map 100% reduce 100%
12/10/26 23:23:17 INFO mapred.JobClient:    Job complete:
job_201210262315_0002
12/10/26 23:23:17 INFO mapred.JobClient:    Counters: 17
12/10/26 23:23:17 INFO mapred.JobClient:    Job Counters
12/10/26 23:23:17 INFO mapred.JobClient:    Launched reduce
tasks=1
12/10/26 23:23:17 INFO mapred.JobClient:    Launched map tasks=1
12/10/26 23:23:17 INFO mapred.JobClient:    Data-local map
tasks=1
12/10/26 23:23:17 INFO mapred.JobClient:    FileSystemCounters
12/10/26 23:23:17 INFO mapred.JobClient:    FILE_BYTES_READ=46
12/10/26 23:23:17 INFO mapred.JobClient:    HDFS_BYTES_READ=16
12/10/26 23:23:17 INFO mapred.JobClient:    FILE_BYTES_
WRITTEN=124
12/10/26 23:23:17 INFO mapred.JobClient:    HDFS_BYTES_WRITTEN=24
12/10/26 23:23:17 INFO mapred.JobClient:    Map-Reduce Framework
12/10/26 23:23:17 INFO mapred.JobClient:    Reduce input groups=4
12/10/26 23:23:17 INFO mapred.JobClient:    Combine output
records=4
12/10/26 23:23:17 INFO mapred.JobClient:    Map input records=1
12/10/26 23:23:17 INFO mapred.JobClient:    Reduce shuffle
bytes=46
12/10/26 23:23:17 INFO mapred.JobClient:    Reduce output
records=4
12/10/26 23:23:17 INFO mapred.JobClient:    Spilled Records=8
12/10/26 23:23:17 INFO mapred.JobClient:    Map output bytes=32
12/10/26 23:23:17 INFO mapred.JobClient:    Combine input
records=4
12/10/26 23:23:17 INFO mapred.JobClient:    Map output records=4
12/10/26 23:23:17 INFO mapred.JobClient:    Reduce input
records=4
```

(3) 执行以下命令：

```
$ hadoop fs -ls out
Found 2 items
drwxr-xr-x    - hadoop supergroup    0 2012-10-26 23:22 /
user/hadoop/out/_logs
-rw-r--r--    1 hadoop supergroup    24 2012-10-26 23:23 /
user/hadoop/out/part-r-00000
```

(4) 现在执行这个命令：

```
$ hadoop fs -cat out/part-0-00000
This    1
a  1
is    1
test.    1
```

原理分析

刚才，我们做了如下3件事情：

- 将之前创建的文本文件移到HDFS的一个新路径下；
- 以上述新路径和一个不存在的输出路径为参数，运行WordCount例程；
- 使用fs程序检查MapReduce作业的输出。

如前所述，伪分布式模式有着更多的Java进程，那么字数统计作业的输出明显短于独立模式下计算圆周率的作业，这点看似奇怪。原因在于，本地独立模式将每个单独任务执行的信息都打印在屏幕上，而在其他模式下，这些信息只被写入运行主机的日志文件中。

输出路径由Hadoop自己创建，实际的结果文件遵守part-nnnn的约定。虽然由于我们设置的原因，仅有一个结果文件。我们使用fs -cat命令来检查文件，结果和预期一致。



如果指定一个已有目录作为Hadoop作业的输出路径，作业将无法运行，并会抛出异常抗议一个已经存在的目录。如果想让Hadoop将输出存储到一个目录，它必须是不存在的目录。把这个特点当做Hadoop的一种安全机制，它可以防止Hadoop重写有用的文件以及用户总是忘记弄清的事。后面将会讲到，如果用户有足够信心，也可以覆盖这种方式。

圆周率和字数统计程序只是Hadoop附带例子的一小部分。下面是如何获得全部例子列表的方法。看看你能否理解其中部分内容。

```
$ hadoop jar hadoop/hadoop-examples-1.0.4.jar
```

一展身手：在更大规模的文本上进行字数统计

运行如此复杂的Hadoop框架，利用5个分立的Java进程来统计一个单行文本文件的字数，并不会给人留下非常深刻的印象。Hadoop令人震惊的力量源自一个事实，即我们可以使用完全相同的程序对一个较大的文件进行字数统计，甚至是分布在多个节点组成的Hadoop集群上的庞大语料库文本。处理这种任务时，执行的命令和刚才完全相同：运行字数统计程序并指定源数据和输出数据目录的位置。

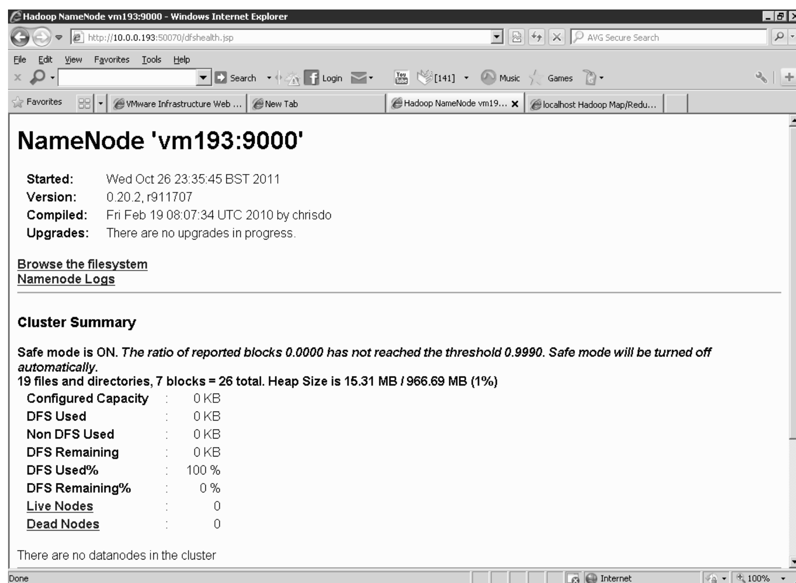
找一个大型在线文本文件，位于http://www.gutenberg.org的Gutenberg项目就是一个很好的案例，把它拷贝到HDFS并执行WordCount例程来对它进行字数统计。输出可能会与您的预期有所不同，因为在一大段文字中，脏数据、标点符号和格式问题都需要解决。想想应当如何改进WordCount，我们将在下一章研究如何将其扩展到一个更复杂的处理链。

通过浏览器查看Hadoop活动

截至目前，我们一直依靠命令行工具和直接的命令输出来观察系统的运行状况。Hadoop提供了两个你应当熟练使用的网络接口：一个用于HDFS，另一个用于MapReduce。两者对伪分布式Hadoop和全分布式Hadoop意义重大，尤其是对全分布式Hadoop至关重要。

HDFS网络用户接口

将浏览器指向运行着Hadoop主机的50030端口。默认情况下，Web界面对于本地主机和具有网络访问权限的任何其他机器都是可用的。下面是一个截图示例：

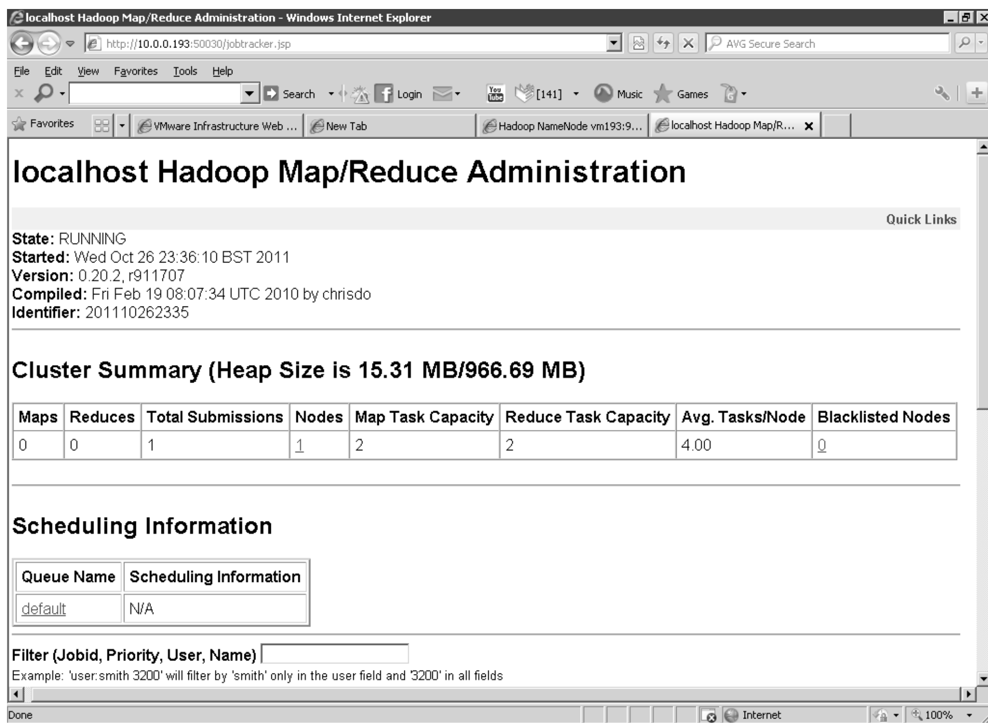


这个接口包含了很多信息，但我们可以通过一些关键数据直观地获悉集群的节点数、文件系统的大小、已用空间以及用于获取更多信息甚至浏览整个文件系统的链接。

读者需要花些时间来慢慢熟悉这个接口，这是很有必要的。在一个多节点的集群中，活跃节点和死亡节点的信息，以及这些节点的详细的历史状态信息对于调试集群故障起着关键作用。

- MapReduce的网络用户接口

JobTracker网络用户接口的默认端口是50070，前面讲的访问规则同样适用于此。下面是一个截图示例：



它比HDFS接口更为复杂！除了类似的活跃节点、死亡节点数量外，它还提供了启动以来已执行的作业数，以及每个作业拆分的任务数。

正在执行的作业和已完成作业列表通常能够提供更多信息。对每个作业来讲，我们可以访问每个任务在每个节点上的运行情况，可以访问包含更详细信息的日志。现在，我们将揭示一个任何分布式系统都存在的令人非常头疼的问题：调试。在分布式系统上进行调试真的是非常困难。

假设你试图使用100台机器组成的集群处理巨大的数据集，其中每台主机都要执行数百个map和reduce任务。如果作业开始缓慢地运行或出现了明显的失败，问题出在什么地方通常并不明显。查看MapReduce的网页用户接口很可能是诊断问题的第一站，因为该接口提供的丰富信息是调查正在运行的作业和已完成作业的入手点。

2.12 使用弹性 MapReduce

现在，我们转去云端Hadoop——由Amazon Web Services提供的弹性MapReduce服务。有多种访问EMR的方法，但现在，我们会专注于亚马逊提供的Web控制台。Web控制台使用完全点击式的Hadoop操作方法，与之前使用命令行的操作方式形成了鲜明对比。

创建Amazon Web Services账号

在使用弹性MapReduce之前，需要创建一个Amazon Web Services账号并用它注册必需的服务。

1. 创建AWS账号

Amazon通用账号已经集成了AWS服务，也就是说，如果读者已经有了任意一个Amazon零售网站的账号，用它即可访问AWS服务。

需要注意的是，AWS服务是付费服务，用户需要将有效的信用卡与其Amazon账号绑定，费用将由此信用卡支付。

如果读者需要一个新的Amazon账户，访问http://aws.amazon.com，选择“create a new AWS account”，并按照提示操作。Amazon已经增加了一个免费服务层，所以读者可能发现，在初期的测试和探索阶段，许多活动都是在非付费层进行的。免费层的范围已经扩大，所以要确信你知道哪些是收费的以及哪些是免费的。

2. 注册必需的服务

有了一个Amazon账号之后，需要用它注册要用到的AWS服务：简单存储服务（S3）、弹性计算云（EC2）和弹性MapReduce（EMR）。单纯注册任何AWS服务都是不收费的，这个过程只是让您的账号可以使用这些服务。

从http://aws.amazon.com页面的链接转到S3、EC2、EMR页面，并点击每个页面上的“Sign up”按钮，然后按照提示操作。



注意！这些付费服务花掉的是真金白银！

在进一步讨论之前，重要的是要明白：使用AWS服务将产生费用，这些费用将出现在您的Amazon账号关联的信用卡。大部分费用是很少的，它会随着消耗的基础设施量而增加。在S3，存储10 GB数据的费用是存储1 GB数据的10倍以上；运行20个EC2实例的费用和将一个实例运行20次的费用相当。Amazon有一个层次收费模型，所以实际成本趋向在较高层次产生较小的边际增幅。但是，你应该在使用任何服务之前，仔细阅读每个服务的定价部分。还请注意，目前将数据从AWS服务（如EC2和S3）移出是收费的，而在这些服务之间转移数据是不收费的。这就意味着，认真设计AWS的使用，在尽可能多的数据处理环节将数据保留在AWS，往往是最具成本效益的。

2.13　实践环节：使用管理控制台在 EMR 运行 WordCount

让我们直接跳到EMR的例子，它使用了一些自带的示例代码。执行下列步骤。

(1) 浏览网页http://aws.amazon.com，访问**Developers | AWS Management Console**，然后点击**Sign in to the AWS Console**按钮。默认视图应该与下图类似。否则，点击**Amazon S3**。



(2) 如以上截图所示，点击“Create bucket”按钮并输入新建桶的名称。桶的名称对所有AWS用户都是全局唯一的，因此一些很明显的名称必然是不可用的，如`mybucket`或`s3test`。

(3) 点击**Region**下拉菜单并选择最近的地理区域。

Create a Bucket - Select a Bucket Name and Region　Cancel

A bucket is a container for objects stored in Amazon S3. When creating a bucket, you can choose a Region to optimize for latency, minimize costs, or address regulatory requirements. For more information regarding bucket naming conventions, please visit the Amazon S3 documentation.

Bucket Name: garryt1use
Region: US Standard

Set Up Logging >　Create　Cancel

(4) 点击**Elastic MapReduce**链接并点击**Create a new Job Flow**按钮。你将看到类似下列截图的页面。

Create a New Job Flow　Cancel

DEFINE JOB FLOW　SPECIFY PARAMETERS　CONFIGURE EC2 INSTANCES　ADVANCED OPTIONS　BOOTSTRAP ACTIONS　REVIEW

Creating a job flow to process your data using Amazon Elastic MapReduce is simple and quick. Let's begin by giving your job flow a name and selecting its type. If you don't already have an application you'd like to run on Amazon Elastic MapReduce, samples are available to help you get started.

Job Flow Name*: My Job Flow

Job Flow Name doesn't need to be unique. We suggest you give it a descriptive name.

Create a Job Flow*: Run your own application
Run a sample application
Choose a Job Type

Run your own application: specify your own parameters for your applications using Hive Program, Custom JAR, Streaming or Pig Program

Run a sample application: by selecting a sample application, parameters will be filled with the necessary data to create a sample Job Flow.

* Required field

Continue

(5) 现在，应该看到类似上述截图的页面。选择**Run a sample application**单选按钮，从示例程序下拉框选择**Word Count (Streaming)**菜单，点击**Continue**按钮。

Create a New Job Flow　Cancel

DEFINE JOB FLOW　SPECIFY PARAMETERS　CONFIGURE EC2 INSTANCES　ADVANCED OPTIONS　BOOTSTRAP ACTIONS　REVIEW

Specify Mapper and Reducer functions to run within the Job Flow. The mapper and reducers may be either (i) class names referring to a mapper or reducer class in Hadoop or (ii) locations in Amazon S3. (Click Here for a list of available tools to help you upload and download files from Amazon S3.) The format for specifying a location in Amazon S3 is bucket_name/path_name. The location should point to an executable program, for example a python program. Extra arguments are passed to the Hadoop streaming program and can specify things such as additional files to be loaded into the distributed cache.

Input Location*: elasticmapreduce/samples/wordcount/input
The URL of the Amazon S3 Bucket that contains the input files.

Output Location*: garryt1use/wordcount/output/2011-11-02
The URL of the Amazon S3 Bucket to store output files. Should be unique.

Mapper*: elasticmapreduce/samples/wordcount/wordSplitter.py
The mapper Amazon S3 location or streaming command to execute.

Reducer*: aggregate
The reducer Amazon S3 location or streaming command to execute.

Extra Args:

‹ Back　Continue　* Required field

(6) 如以上截图所示，下一步的屏幕允许用户指定作业的输出位置。在输出位置文本框中，输入步骤1创建的桶的名称（这里使用的桶的名称为garryt1use），然后点击**Continue**按钮。

Create a New Job Flow　Cancel

DEFINE JOB FLOW　SPECIFY PARAMETERS　CONFIGURE EC2 INSTANCES　ADVANCED OPTIONS　BOOTSTRAP ACTIONS　REVIEW

Specify the Master, Core and Task Nodes to run your job flow. For more than 20 instances, complete the limit request form.

Master Instance Group: This EC2 instance assigns Hadoop tasks to Core and Task Nodes and monitors their status.

Instance Type: Small (m1.small)　Request Spot Instance

Core Instance Group: These EC2 instances run Hadoop tasks and store data using the Hadoop Distributed File System (HDFS). Recommended for capacity needed for the life of your job flow.

Instance Count: 2
Instance Type: Small (m1.small)　Request Spot Instances

Task Instance Group (Optional): These EC2 instances run Hadoop tasks, but do not persist data. Recommended for capacity needed on a temporary basis.

Instance Count: 0
Instance Type: Small (m1.small)　Request Spot Instances

‹ Back　Continue　* Required field

(7) 在下一步的页面中，用户可以修改作业所用虚拟主机数及其规格。确保每个组合框的实例类型为**Small (m1.small)**，核心组的节点数量为**2**，任务组的节点数量为**0**。然后点击**Continue**按钮。

Create a New Job Flow
Cancel

DEFINE JOB FLOW SPECIFY PARAMETERS CONFIGURE EC2 INSTANCES ADVANCED OPTIONS BOOTSTRAP ACTIONS REVIEW

Here you can select an EC2 key pair, set your job flow debugging options, and enter advanced job flow details such as whether it is a long running cluster.

Amazon EC2 Key Pair: Proceed without an EC2 Key Pair
Use an existing Key Pair to SSH into the master node of the Amazon EC2 cluster as the user "hadoop".

Configure your debugging options. Learn more.

Enable Debugging: Yes No
Amazon S3 Log Path:
An Amazon S3 Log Path is required if you are enabling debugging.

Enable Hadoop Debugging: Yes No
To enable Hadoop Debugging you will need to sign up for Amazon SimpleDB.

Set advanced job flow options.

Keep Alive Yes No This job flow will run until manually terminated.

‹ Back Continue * Required field

(8)“下一步”的截图包含了一些本例中用不到的一些选项。对于**Amazon EC2 key pair**区域，选择**Proceed without key pair**菜单，对于**Enable Debugging**区域，选择**No**。确保**Keep Alive**单选按钮被设为**No**并单击**Continue**按钮。

Create a New Job Flow
Cancel

DEFINE JOB FLOW SPECIFY PARAMETERS CONFIGURE EC2 INSTANCES ADVANCED OPTIONS BOOTSTRAP ACTIONS REVIEW

Proceed with no Bootstrap Actions
I do not want to associate any Bootstrap Actions with this Job Flow.

NOTE: Bootstrap Actions must be associated with a Job Flow upon creation. You will not be able to add these later without creating a new Job Flow.

Configure your Bootstrap Actions

‹ Back Continue * Required field

(9) 如上图所示，用户现在并不需要在“下一步”页面做太多工作。确认**Proceed with no Bootstrap Actions**单选按钮被选中并点击**Continue**按钮。

Create a New Job Flow　　Cancel

DEFINE JOB FLOW　SPECIFY PARAMETERS　CONFIGURE EC2 INSTANCES　ADVANCED OPTIONS　BOOTSTRAP ACTIONS　REVIEW

Please review the details of your job flow and click "Create Job Flow" when you are ready to launch your Hadoop Cluster.

Job Flow Name: My Job Flow
Type: Word Count (Streaming)
Edit Job Flow Definition

Input Location: s3n://elasticmapreduce/samples/wordcount/input
Output Location: s3n://garryt1use/wordcount/output/2011-11-02
Mapper: s3n://elasticmapreduce/samples/wordcount/wordSplitter.py
Reducer: aggregate
Extra Args:
Edit Job Flow Parameters

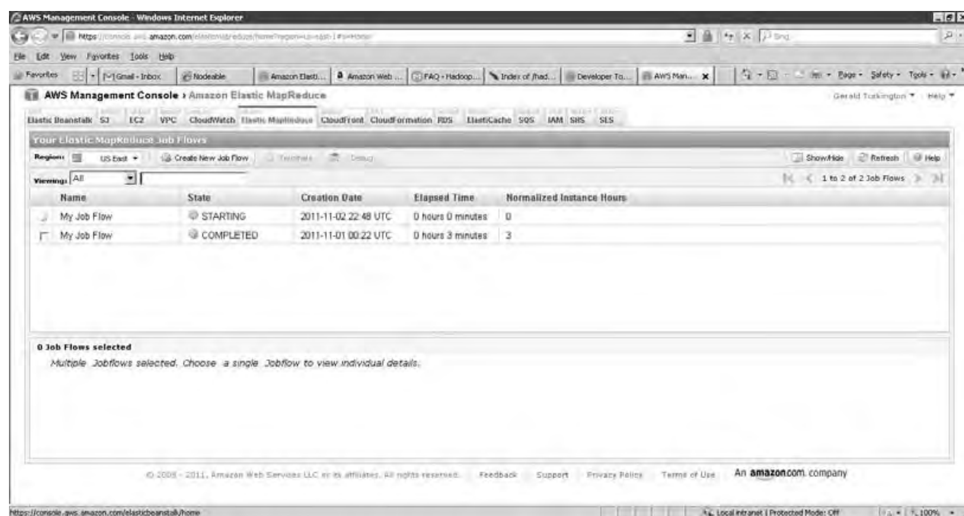
Master Instance Type: m1.small　**Instance Count:** 1
Core Instance Type: m1.small　**Instance Count:** 2
Edit EC2 Configs

Amazon EC2 Key Pair:
Amazon S3 Log Path:
Enable Hadoop Debugging: No
Keep Alive: No
Edit Advanced Options

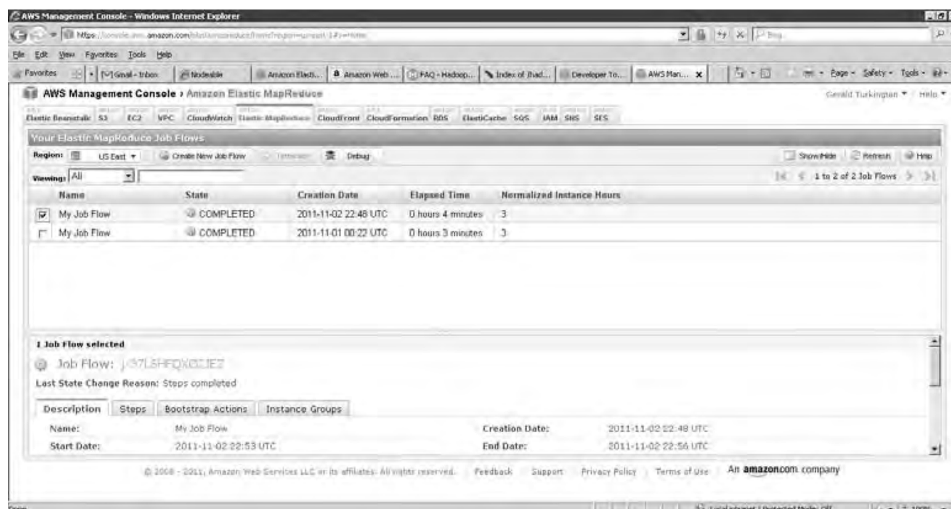
Bootstrap Actions: No Bootstrap Actions created for this Job Flow
Edit Bootstrap Actions

‹ Back　　Create Job Flow　　**Note:** Once you click "Create Job Flow," instances will be launched and you will be charged accordingly.

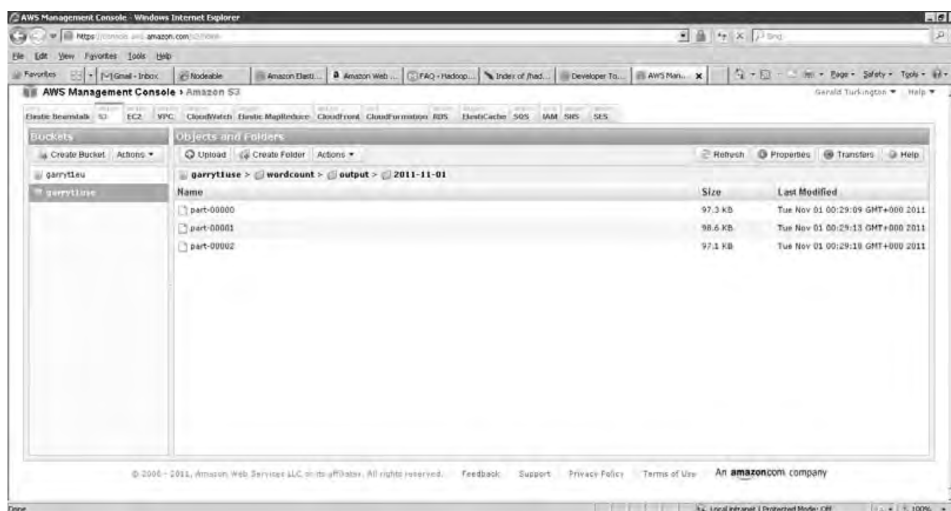
(10) 确认作业流的设置与预期一致，点击**Create Job Flow**按钮。之后点击**View my Job Flows**和**check status**按钮。这样会列出用户的所有作业流，用户可筛选只显示正在运行的作业或已结束的作业。默认设置是显示所有作业，如同下图所示。



(11) 不时地点击**Refresh**按钮，直到列出的作业状态从**Running**或**Starting**变为**Complete**，然后点击其复选框查看作业流的详情，如下图所示。



(12) 点击**S3**标签并选择你为输出位置所创建的容器。你将看到它有一个称为**wordcount**的入口，这是一个路径。右击它并选择**Open**。然后重复上述动作直至看到真实文件的列表，它们遵守类似Hadoop的part-*nnnnn*命名规则，如下图所示。



右击并打开**part-00000**。它应该看起来与下列内容类似：

```
a        14716
aa       52
aakar    3
```

```
aargau          3
abad            3
abandoned       46
abandonment     6
abate           9
abauj           3
abbassid        4
abbes           3
abbl            3
...
```

这种类型的输出看起来是不是有点熟悉？

原理分析

第1个步骤是针对S3进行的，而非EMR。S3是一个可扩展的存储服务，它允许用户在叫做桶的容器内存储文件（称为对象），并使用桶和对象的键（也就是名称）来访问对象。该模型类似于文件系统的使用情况，虽然也有潜在的差异，但这些区别在本书中并不重要。

S3是放置MapReduce程序及待处理的源数据的地方，也是存储输出数据及EMR Hadoop作业日志的地方。有大量的第三方工具可以访问S3，但在这里我们使用AWS管理控制台，它是访问大多数AWS服务的浏览器界面。

虽然我们建议用户为S3选择最近的地理区域，但这不是必需的。美国之外的地区，Amazon通常会为接近它们的客户提供较小的延迟，但用户也往往需要支付略高的费用。在什么地方托管数据和应用程序，需要用户综合考虑所有因素后决定。

创建S3存储桶后，我们转到EMR控制台并创建了一个新的作业流。在EMR，这个术语用来指代一个数据处理任务。正如我们将要看到的，它可以是一个一次性作业，底层的Hadoop集群按需创建和销毁；也可以是一个长期运行的集群，在它上面执行着多个作业。

我们保留了了默认的作业流名称，然后选择使用一个示例应用程序。在这个案例中，示例程序是用Python实现的字数统计程序WordCount。Hadoop Streaming是指一种机制，它允许使用脚本语言编写map和reduce任务，但功能与我们先前使用的Java字数统计程序相同。

配置作业流的表单需要填写源数据和程序的位置，map和reduce类的位置及所需的输出数据的位置。对于我们刚刚看到的这个例子，大多数区域都是预先填好的，可以看出，这与在命令行中运行本地Hadoop所需的内容，有明显的相似之处。

通过不选择**Keep Alive**选项，我们创建了一个一次性Hadoop集群，它专门为执行本作业而创建，作业完成后就会被销毁。这类集群的启动时间较长，但会最大限度地降低成本。如果您选择保持作业流活跃，你会看到更多的作业被更迅速地执行，因为你不必等待集群启动。但是你需要

为EC2基础资源支付费用，直到明确地终止该作业流。

在确认之后，我们不需要添加任何附加的启动选项。我们选择了希望部署在Hadoop集群的主机类型和主机数。EMR有3组不同的主机。

- **主实例组**：这是一个承载着NameNode和JobTracker的控制节点。主实例组中只有1个这样的节点。
- **核心实例组**：这些实例是运行着HDFS DataNode和MapReduce TaskTracker的节点。这些主机的数量是可配置的。
- **任务实例组**：这些主机不持有HDFS数据，却运行TaskTracker，并能提供更多的处理能力。主机的数量是可配置的。

主机类型代表着不同级别的硬件能力，详细内容可在EC2页面上找到。较大的主机有较强的运算能力，同时价格也较高。目前，默认情况下，一个作业流的主机总数一定小于等于20个，但亚马逊有一种简单的形式可以申请更高的限制。

确认后，一切都如预期，我们启动作业流并在控制台上查看其运行状况，直到状态变为**COMPLETED**。这时，我们回到S3，查看由我们指定的作为输出目标的桶，并检查字数统计作业的输出。它应当与本地Hadoop的字数统计的输出看起来非常相似。

一个明显的问题是，源数据来自哪里？这是作业流设置中的一个预填充字段，我们在创建作业流的过程中看到过。对于非持久性的作业流，最常见的模型是，从一个指定的S3源位置读取数据并将所得到的数据写入到指定的结果S3桶中。

这就是AWS管理控制台！它允许用户通过浏览器对服务（如S3和EMR）进行细粒度的控制。仅需拥有一个浏览器和一张信用卡，我们就可启动Hadoop作业来处理数据，无需担心安装、运行、管理Hadoop的任何机制问题。

一展身手：其余EMR示例程序

EMR提供了其他几个示例应用程序，为什么不同时试试它们呢？

2.13.1 使用EMR的其他方式

虽然AWS管理控制台是一个强大的和令人印象深刻的工具，它并不总是我们想要用来访问S3和运行EMR作业的方式。如同所有的AWS服务，用户可通过可编程的工具或命令行工具来使用这些服务。

1. AWS证书

然而，在使用可编程工具或命令行工具之前，我们需要明白账号持有人如何通过AWS，从

而提出服务申请。因为这些都是收费服务，用户确实不希望其他任何人以他们的名义提出服务申请。请注意，在之前的例子中，我们用自己的AWS账号直接登录到AWS管理控制台，因此无需担心这个问题。

每个AWS账号都有若干个标识符，这些标识符会在访问这些服务的时候用到。

- **账号ID**：每个AWS账号都有一个数字ID。
- **访问秘钥**：每个账号都有一个关联的访问秘钥，它用于申请服务时的账号验证。
- **秘密访问秘钥**：秘密访问秘钥是访问秘钥的搭档。访问秘钥并不是私密的，它会暴露在服务请求过程中，但是秘密访问秘钥只由账号主人保管，可用于证明账号主人的身份。
- **密钥对**：这是用于登录到EC2主机的密钥对。既可以在EC2中生成公钥/私钥密钥对，也可以将外部生成的密钥对导入到系统中。

似乎听起来有点乱，实际情况确实如此——至少第一次接触时是这样的。然而，当使用工具访问AWS服务时，通常只要事先将正确的证书加入配置文件，一切都会正常工作。但是，如果读者决定深入研究可编程工具或命令行工具，值得投入一点时间来阅读每个服务的文档，以了解其安全机制的工作原理。

2. EMR命令行工具

本书中，我们不会用S3和EMR来实现任何不能从AWS管理控制台实现的事情。然而，当处理业务工作负载，整合其他工作流程或自动化访问服务时，无论基于浏览器的工具有多强大，它都不适用于这样的问题。使用服务的直接编程接口可以实现最精细的控制，但却需要付出最多的努力。

Amazon为许多服务提供了一组命令行工具，这是一种自动访问AWS服务的有效方式，可以最大限度地减少所需的开发工作量。如果读者想再实现一个基于CLI的EMR接口，却又不想编写自定义代码的话，可以试试链接于EMR主页面的弹性MapReduce命令行工具。

2.13.2　AWS生态系统

每种AWS服务都有大量的第三方工具、服务和函数库，它们可以提供访问服务的不同方式、额外的功能，或新的实用程序。作为熟悉AWS生态系统的起点，请到http://aws.amazon.com/developertools页面查看开发人员工具集。

2.14　本地 Hadoop 与 EMR Hadoop 的对比

有了关于本地Hadoop集群及EMR Hadoop集群的第一次经验后，这是一个考虑这两种方法差异的好时机。

二者的差异可能是明显的，其关键区别并不在于能力。如果我们需要的是一个运行MapReduce作业的环境，任何一种方法都没问题。相反，主要的区别点在第1章中曾提及的一个话题，那就是：你是喜欢一个包含前期基础设施费用以及持续的维护工作的成本模型，还是喜欢一个具有较低维护负担、可快速实现的、理论上具有无限扩展性的按需付费的成本模型。除了成本因素外，要牢记以下几点内容。

- EMR支持特定版本的Hadoop，并会随着时间推移升级Hadoop版本。如果用户需要某个特定的版本，尤其是，如果用户需要在新版本发布后马上获得最新、最稳定的版本，那么EMR Hadoop升级到所需版本之前的时间间隔是用户不可接受的。
- 用户可以启动一个持久的EMR作业流，并将其视为一个本地Hadoop集群，登录到主机节点，并调整它们的配置。如果你发现自己正在这样做，值得好好考虑一下，果真需要这样控制Hadoop集群吗？如果答案是肯定的，那么从本地Hadoop转到EMR带来的成本优势是否依然存在？
- 如果它最终归结为成本问题，切记要考虑本地集群的所有隐性成本，人们通常会忘掉这些隐性成本。想想电力、场地、制冷和主机设备的费用。且不说管理开销，如果设备在凌晨出现故障，管理成本也是一笔不小的开销。

2.15 小结

本章内容讲述了如何安装并运行一个Hadoop集群，以及在Hadoop集群上执行MapReduce程序的步骤。

具体来说，我们讨论了在本地Ubuntu主机上运行Hadoop的前提条件，也介绍了如何以独立式或伪分布式模式安装和配置本地Hadoop集群。之后，我们讲解了如何访问HDFS文件系统和提交MapReduce作业。然后，我们继续前进，学习了访问弹性MapReduce和其他AWS服务需要用到的账号。

我们了解了如何使用AWS管理控制台浏览并创建S3桶和对象，以及如何创建一个作业流并用它在EMR托管的Hadoop集群上执行MapReduce作业。我们还讨论了访问AWS服务的其他方式，并研究了本地Hadoop和EMR托管Hadoop之间的差异。

既然我们已经学习了在本地或EMR运行Hadoop的方法，我们已经准备好开始编写自己的MapReduce程序了，这将是下一章的主题。

第3章

理解MapReduce

前两章讨论了Hadoop可以解决的问题，并有了一些运行MapReduce示例作业的实践经验。在此基础上，接下来我们继续深入研究。

通过本章学习，读者将会：

- 理解键值对为何可以成为Hadoop任务的基础；
- 了解MapReduce作业的多个阶段；
- 详细探讨map、reduce以及可选组合阶段的工作原理；
- 学习Hadoop Java API，并用它开发一些简单的MapReduce作业；
- 了解Hadoop的输入和输出。

3.1 键值对

自第1章开始，我们一直在谈论以键值对的形式处理数据并输出结果，而没有解释为什么要以键值对的形式进行。现在是时候来阐述这个问题了。

3.1.1 具体含义

首先，我们会通过强调Java标准库中的类似概念，来阐明我们所说的键值对的含义。`java.util.Map`接口是常用类，如`HashMap`，甚至原始`Hashtable`的父类（通过向后重构代码库）。

对于任何Java `Map`对象，其内容是从指定类型的给定键到相关值的一组映射，键与值的数据类型可能不同。例如，一个`HashMap`对象可以包含从人名（`String`）到其生日（`Date`）的一组映射。

Hadoop中的数据包含与相关值关联的键。这些数据的存储方式允许对数据集的不同值根据键进行分类和重排。如果使用键值对数据，应该会有如下疑问：

- ❑ 在数据集中，一个给定的键必然有映射值吗？
- ❑ 给定键的关联值是什么？
- ❑ 键的完整集合是什么？

回忆一下前一章提到的WordCount。稍后，我们将更详细地讨论它，然而该程序的输出显然是键/值关系的集合。对于每个字（键），都有对应着它出现的次数（值）。琢磨一下这个简单的例子，键/值数据的一些重要特征就变得清晰起来，具体如下：

- ❑ 键必须是唯一的，而值并不一定是唯一的；
- ❑ 每个值必须与键相关联，但键可能没有值（虽然在这个特定的例子中没有出现这种情况）；
- ❑ 对键进行明确定义非常重要。它决定了计数是否区分大小写，这将产生不同的结果。



请注意，我们需要审慎对待“键是唯一的”这一概念。这并不是说键只出现一次。在我们的数据集中，可以看到键多次出现。并且我们将看到，MapReduce模型有一个将所有与特定键关联的数据汇集的步骤。键的唯一性保证了，假如我们为某一给定键汇集对应的值，结果将是从该键的实例到每个值的映射，不会忽略掉任何值。

3.1.2 为什么采用键/值数据

键/值数据作为MapReduce操作的基础，成就了一个强大的编程模型，使MapReduce获得了令人惊讶的广泛应用。Hadoop和MapReduce被多种不同行业和问题领域所采用即证实了这一点。很多数据要么本身即为键/值形式，要么可以以键/值这种方式来表示。键值数据这一简单的模型具有广泛的适用性，以这种形式定义的程序可以应用于Hadoop框架。

当然，数据模型本身并非是使Hadoop如此强大的唯一要素，它真正的强大之处在于如何运用并行处理技术以及分而治之思想，这些都在第1章中讨论过。我们可以在大量主机上储存、执行数据，甚至使用将较大任务分割成较小任务的框架，然后将所有的并行结果整合成最终结论。但是，我们需要上述框架提供一种描述问题的方法，即便用户不懂该框架的运行机理，也能表述清楚要处理的问题。我们只需对数据所需的转换进行描述，其余事情由该框架完成。MapReduce利用其键/值接口提供了这样的抽象：程序员只需指定所要求的转换，Hadoop完成对任意规模数据集的复杂的数据转换处理过程。

一些实际应用

为了更为具体地理解键值对，可以想像一些实际应用的键值对数据：

- ❑ 通讯簿将一个名字（键）和联系方式（值）关联起来；

- ❑ 银行帐号使用一个帐号（键）关联帐户明细（值）;
- ❑ 一本书的索引关联一个关键字（键）和其所在页码（值）;
- ❑ 在计算机文件系统中，根据文件名（键）访问各类数据，如文本、图片和语音（值）。

我们刻意列举了一些范围宽泛的例子，帮助读者认识到，键/值数据并不是只能应用于高端数据挖掘的约束模型，而是环绕我们身边的非常普通的模型。

我们之所以大篇幅地讨论键值对数据，是因为深入理解键值对的概念对理解并使用Hadoop非常重要。MapReduce只能处理以键值对形式描述的数据。

3.1.3 MapReduce作为一系列键/值变换

读者可能偶然见过把MapReduce描述成一系列键/值转换的描述方法。这种描述方法看上去有点吓人。

```
{K1,V1} -> {K2, List<V2>} -> {K3,V3}
```

现在我们试图去理解上式所表达的意思。

- ❑ MapReduce作业的`map`方法的输入是一系列键值对，称之为`K1`和`V1`。
- ❑ `map`方法的输出（今后作为`reduce`方法的输入）是一系列键以及与之关联的值列表，称之为`K2`和`V2`。需要注意的是，每个mapper仅仅输出一系列单个的键值对，它们通过`shuffle`方法组合成键与值列表。
- ❑ MapReduce作业的最终输出是另一串键值对，称之为`K3`和`V3`。

这些键值对的集合可能相同也可能不同，也就是说，很可能输入姓名和联系方式然后输出相同内容，也许附带有一些用于整理信息的中间格式。请在接下来探索MapReduce的Java API时记住这个3阶段模型。首先，我们会浏览将要用到的API的主要部分，然后系统剖析一个MapReduce作业的执行过程。

随堂测验：键值对

问题1 键值对的概念是什么？

(1) Hadoop创造并专用的概念。

(2) 表述我们经常看到但并没有这样考虑的事物间关系的一种方法。

(3) 一个计算机科学的学术概念。

问题2 用户名/密码组合是键值对的一个例子吗？

(1) 是的，这是一个值与另一个值相关联的明显例子。

(2) 不是，密码更像是用户名的一个属性，它们之间没有索引–类型的关系。

(3) 通常我们并不这样认为，但是Hadoop仍可将用户名/密码组合作为键值对组合来处理。

3.2 MapReduce 的 Hadoop Java API

Hadoop API在0.20版发生了重大变化，构成了本书使用的Hadoop 1.0版的主要接口。虽然之前的API功能运转正常，但社会各界认为，它在某些方面是笨拙的且被弄得不必要的复杂。

新API，有时也称为上下文对象，其原因我们将在后面看到，它是使用Java开发MapReduce的未来趋势。因此，本书将尽量使用新API。需要注意的是：部分0.20之前的MapReduce库没有被移植到新API，所以当我们需要分析其中任何接口时，将使用旧API。

0.20 MapReduce Java API

在`org.apache.hadoop.mapreduce`包或其子包中，包含了0.20及以上版本的MapReduce API实现的大部分关键类和接口。

`org.apache.hadoop.mapreduce`包提供了`Mapper`和`Reducer`基类。在大多数情况下，一个MapReduce作业会继承上述基类，实现针对该作业的`Mapper`和`Reducer`子类。



虽然最近的Hadoop API使用了`KEYIN/VALUEIN`和`KEYOUT/VALUEOUT`之类的术语，但由于`K1/K2/K3/`等术语有助于理解端到端的数据流，本书将坚持使用`K1/K2/K3/`等形式的术语。

1. **`Mapper`类**

这是Hadoop提供的`Mapper`基类的缩减视图。自定义的mapper类的实现将继承该基类并重写指定的方法，如下所示：

```
class Mapper<K1, V1, K2, V2>
{
      void map(K1 key, V1 value Mapper.Context context)
            throws IOException, InterruptedException
{..}
}
```

虽然使用Java泛型使该类乍看起来有些难以理解，其实没那么复杂。该类以键/值数据作为输入和输出类型，其map方法以输入的键值对作为参数。另一个参数是`Context`类的实例，它提供了与Hadoop框架通信的多种机制，其中之一是输出`map`或`reduce`方法的结果。

有时，用户可能需要重写另外3个方法。

```
protected void setup( Mapper.Context context)
      throws IOException, Interrupted Exception
```



请注意，map方法只引用了K1和V1键值对的一个实例。这是MapReduce范式的一个关键特点，在这个范式下，用户编写处理单条记录的类，框架负责将巨大的数据集转化为键值对流的所有工作。读者永远不必编写处理完整数据集的map或reduce类。Hadoop也通过InputFormat和OutputFormat类提供了类似机制，这些类提供了普通文件格式的实现，除特殊文件类型外，用户无需编写文件解析器。

该方法在任何键值对被提交给map方法之前，被调用一次。该方法的默认实现不执行任何操作。

```
protected void cleanup( Mapper.Context context)
      throws IOException, Interrupted Exception
```

该方法在所有键值对被提交给map方法之后，被调用一次。该方法的默认实现不执行任何操作。

```
protected void run( Mapper.Context context)
      throws IOException, Interrupted Exception
```

此方法控制JVM中任务处理的总体流程。该方法的默认实现将在对数据分块中的每个键值对反复调用map方法之前，调用一次setup方法，并最终调用一次cleanup方法。



下载示例代码

读者可以使用http://www.packtpub.com的账号下载所购买的所有Packt书籍的示例代码文件。或者访问http://www.packtpub.com/support，并在该页面注册，读者将收到通过电子邮件寄来的示例代码文件。

2. Reducer类

Reducer基类的运行与Mapper基类非常相似，通常子类仅需重写一个reduce方法。Reducer基类的缩略定义如下：

```
public class Reducer<K2, V2, K3, V3>
{
void reduce(K1 key, Iterable<V2> values,
     Reducer.Context context)
       throws IOException, InterruptedException
{..}
}
```

再次注意，该类以更广泛的数据流的形式进行定义（reduce方法接受K2/V2作为输入，并

提供K3/V3作为输出），而实际reduce方法只需要一个键及与之关联的值列表。同样，Context对象负责输出该方法的结果。

该类也有setup、run和cleanup方法，这些方法的默认实现与Mapper类的类似，可以选择性地进行重写：

```
protected void setup( Reduce.Context context)
throws IOException, InterruptedException
```

该方法在任何键/值列表被提交给reduce方法之前，被调用一次。其默认实现不执行任何操作。


```
protected void cleanup( Reducer.Context context)
throws IOException, InterruptedException
```

该方法在所有键/值列表被提交给reduce方法之后，被调用一次。其默认实现不执行任何操作。

```
protected void run( Reducer.Context context)
throws IOException, InterruptedException
```

该方法对JVM中任务处理的总体流程进行控制。其默认实现在对提供给Reducer类的众多键值对反复调用reduce方法之前，调用setup方法，并在最后调用cleanup方法。

3. **Driver**类

除Mapper和Reducer类外，执行MapReduce作业还需要另外一段代码——负责与Hadoop框架通信并指定运行MapReduce作业所需的配置元素的驱动程序。驱动程序的工作涉及多个方面，例如告诉Hadoop使用哪个Mapper和Reducer类，以何种格式在何处获取输入数据，在何处存放输出数据，以及如何对输出数据进行格式化。驱动程序还负责设置其他各种配置选项，本书将会讲到它们。

作为一个子类，Driver没有默认父类。驱动逻辑通常存在于封装MapReduce作业类的主方法中。下述代码片段是一个示例驱动程序。读者不必纠结于每行代码的工作原理，却要大体明白每行代码实现的配置任务。

```
public class ExampleDriver
{
...
public static void main(String[] args) throws Exception
{
// 创建Configuration对象，用于设置其他选项
    Configuration conf = new Configuration() ;
// 创建作业对象
Job job = new Job(conf, "ExampleJob") ;
// 设置作业jarfile中主类的名字
    job.setJarByClass(ExampleDriver.class) ;
```

```
// 设置mapper类
    job.setMapperClass(ExampleMapper.class) ;
// 设置reducer类
    job.setReducerClass(ExampleReducer.class) ;
// 设置最终输出的键和值的类型
    job.setOutputKeyClass(Text.class) ;
    job.setOutputValueClass(IntWritable.class) ;
// 设置输入和输出文件路径
FileInputFormat.addInputPath(job, new Path(args[0])) ;
FileOutputFormat.setOutputPath(job, new Path(args[1]))
// 执行并等待作业完成
System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}}
```

鉴于之前对作业的讨论，`Driver`类实现的大部分设置与`Job`对象的操作相关，这不足为奇。这些设置包括设置作业的名称，指定用作mapper和reducer的类。

`Driver`类还对某些输入/输出配置进行了设置，最终传递给main方法的参数被用于指定作业的输入和输出位置。这是读者会经常看到的一个很普遍的模式。

对于配置选项，有很多的默认值。在前面的类中，我们隐式地使用其中的一些默认设置。最值得注意的是，我们没有提输入文件的格式及输出文件的写入方式。这些内容在前面提到的`InputFormat`和`OutputFormat`类中进行了定义，我们稍后将对其进行详细探讨。默认的输入和输出格式是文本文件，这些文本文件适用于WordCount示例程序。除特别优化的二进制格式外，有多种表示文本文件内部格式的方法。

对于不太复杂的MapReduce作业，一种常见的模型是将`Mapper`和`Reducer`类作为驱动程序的内置类。这种模型将所有内容都保存在一个文件中，简化了代码部署。

3.3　编写 MapReduce 程序

在相当长的一段时间内，我们一直在使用和谈论WordCount程序。现在，让我们实际编写该程序，编译并运行它，然后尝试进行一些修改。

3.4　实践环节：设置 classpath

为了编译任意Hadoop相关的代码，我们需要参考标准的Hadoop捆绑类。

从软件分发包中添加`Hadoop-1.0.4.core.jar`至Java classpath，如下所示：

```
$ export CLASSPATH=.:${HADOOP_HOME}/Hadoop-1.0.4.core.jar:${CLASSPATH}
```

原理分析

上述命令显式地将`Hadoop-1.0.4.core.jar`文件与当前路径、原本CLASSPATH环境变量的内容添加到classpath。

再次重申，把这个命令放到shell启动文件或一个独立的引用文件将是很好的选择。



稍后，我们还需要将许多用于Hadoop的第三方代码库添加到我们的classpath，而且有一个快捷方式来完成这个工作。就目前而言，明确的将核心JAR文件添加到classpath就足够了。

3

3.5 实践环节：实现 WordCount

在第2章中，我们已经看到了WordCount示例程序的应用。现在，我们将通过执行以下步骤，探索使用Java语言实现这一程序。

(1) 在`WordCount1.java`文件中输入以下代码：

```
import java.io.* ;
import org.apache.hadoop.conf.Configuration ;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;


public class WordCount1
{


    public static class WordCountMapper
    extends Mapper<Object, Text, Text, IntWritable>
{

        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value, Context context
        ) throws IOException, InterruptedException {
            String[] words = value.toString().split(" ") ;

            for (String str: words)
            {
```

```
                    word.set(str);
                    context.write(word, one);
                }
            }
        }

        public static class WordCountReducer
        extends Reducer<Text,IntWritable,Text,IntWritable> {
            public void reduce(Text key, Iterable<IntWritable> values,
                Context context
                ) throws IOException, InterruptedException {
                    int total = 0;
                for (IntWritable val : values) {
                    total++ ;
                }
                context.write(key, new IntWritable(total));
            }
        }

        public static void main(String[] args) throws Exception {
            Configuration conf = new Configuration();
            Job job = new Job(conf, "word count");
            job.setJarByClass(WordCount1.class);
            job.setMapperClass(WordCountMapper.class);
            job.setReducerClass(WordCountReducer.class);
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(IntWritable.class);
            FileInputFormat.addInputPath(job, new Path(args[0]));
            FileOutputFormat.setOutputPath(job, new Path(args[1]));
            System.exit(job.waitForCompletion(true) ? 0 : 1);
          }
        }
```

(2) 现在通过执行下列命令编译`WordCount1.java`:

```
$ javac WordCount1.java
```

原理分析

这是我们实现的第一个完整MapReduce作业。纵观该作业的结构，你应该认出我们先前已经讨论过的要素：在其主方法中通过驱动程序设置的完整`Job`类，以内置类的形式定义的`Mapper`和`Reducer`。

下一节，我们将更详细地探讨MapReduce的机制，但现在通过阅读前述代码思考它是如何实现之前谈到的键/值转换的。

`Mapper`类的输入可以说是最难理解的，因为键并没有得到实际使用。该作业指定TextInput-Format作为输入数据的格式。默认情况下，这种格式提供给mapper的数据中，键指的是文件中的

行号，值指的是该行内容。实际上，读者可能从来没有遇到过使用行号作为键的mapper，但TextInputFormat提供了这样的映射。

对于输入源中的每行文本，mapper都会执行一次，每次运行都会获取该行内容并把它切分成词语。之后，它会使用Context对象以<word,1>的格式输出每个新的键/值（俗称发射）。这些输出就是我们所说的K2/V2值。

之前，我们讲到reducer的输入是一个键和一组与该键对应的值。在map和reduce方法之间发生了一些不可思议的事，将每个键对应的值收集起来，现在我们不描述该过程。Hadoop对每个键执行一次reducer，前面例子中的reducer简单地统计Iterable对象中的数字，并以<word, count>的形式给出每个词的输出。这些输出就是我们所说的K3/V3的值。

总结一下mapper和reducer类的特点：WordCountMapper类以IntWritable和Text作为输入，然后以Text和IntWritable作为输出。WordCountReducer类的输入和输出都是Text和IntWritable类型。这又是一个相当普遍的模式，在该模式中map方法对键和值进行了反转并输出一系列数据对，而reducer对这些数据对进行了整合。

因为有真正的参数值，驱动在这里更有意义。我们使用传给类的参数指定输入和输出位置。

3.6 实践环节：构建 JAR 文件

在Hadoop集群中运行作业之前，我们必须将所需的类文件打包到一个JAR文件，该文件将被提交给系统。

用已生成的类文件创建JAR文件的命令如下所示。

```
$ jar cvf wc1.jar WordCount1*class
```

原理分析

在提交给Hadoop之前，我们必须将类文件打包成JAR文件，无论Hadoop部署在本地还是Elastic MapReduce。



要小心处理JAR命令和文件路径。假如读者在JAR文件中包含了子目录下的文件，该类的存储路径可能与预期不一致。尤其是当所有源数据都在该目录进行编译的时候，这种现象更为普遍。这种情况下，可能需要编写一个脚本来改变路径，将所需文件转换为JAR文件，并将JAR文件转移到所需位置。

3.7　实践环节：在本地 Hadoop 集群运行 WordCount

现在，我们已经生成了类文件并将其打包进了JAR文件，可以通过执行下列步骤运行WordCount。

(1) 提交新JAR文件到Hadoop执行。

```
$ hadoop jar wc1.jar WordCount1 test.txt output
```

(2) 如果执行成功的话，你会看到与前一章我们运行Hadoop提供的示例WordCount相似的输出。检查输出文件，它应当如下所示：

```
$ Hadoop fs -cat output/part-r-00000
This 1
yes 1
a 1
is 2
test 1
this 1
```

原理分析

这是我们首次对自己的代码使用Hadoop JAR命令。它有4个参数：

(1) JAR文件名；

(2) JAR文件中的驱动类名；

(3) 输入文件在HDFS的位置（本例中，是对`/user/Hadoop home`文件夹的相对引用）；

(4) 输出文件夹的目标位置（同样也是一个相对路径）。



只有在JAR文件清单中没有指定主类的情况下，才需要驱动类名（如本例中）。

3.8　实践环节：在 EMR 上运行 WordCount

现在，我们将展示如何在EMR上运行同样的JAR文件。时刻要牢记，这是需要付费的！

(1) 访问AWS控制台http://aws.amazon.com/console，登录并选择S3。

(2) 你将需要两个桶：一个存储JAR文件，另一个存储作业的输出。你既可以使用现有的桶也可以新建两个桶。

(3) 打开将用于存储作业文件的桶，选择上传，并添加之前创建的`wc1.jar`文件。

(4) 返回控制台主页面，之后通过选择**Elastic MapReduce**进入控制台的EMR部分。

(5) 点击**Create a New Job Flow**按钮，你会看到一个如下所示的熟悉界面。



(6) 第2章中，我们使用的是Hadoop自带示例程序，因此选择的是**Run a sample application**。本例中，为了运行自己的代码，我们需要执行不同的步骤。首先，选择**Run your own application**单选按钮。

(7) 在**Select a Job Type**下拉框中，选择**Custom JAR**。

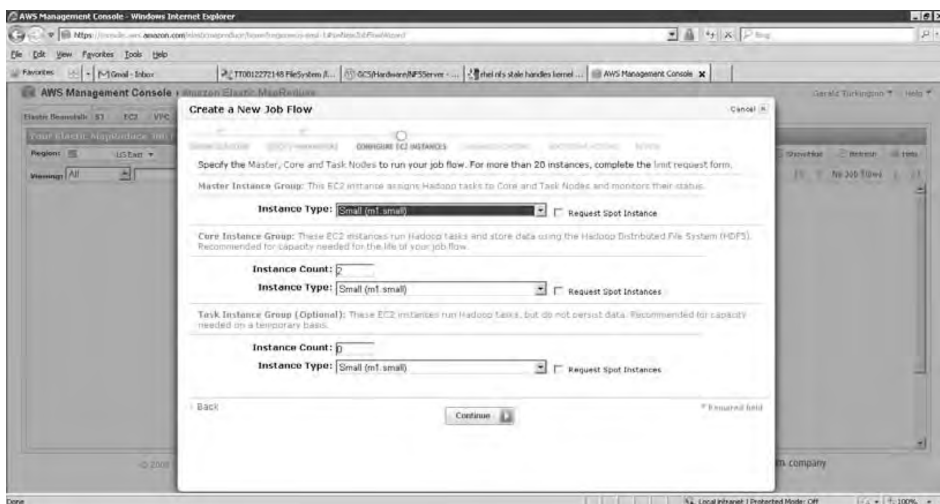
(8) 点击**Continue**按钮，你将看对一个新表格，如下图所示。



现在指定该作业的参数。在我们已上传的JAR文件中，代码（尤其是驱动类）指定了一些参数，例如`Mapper`和`Reducer`类。

我们需要提供的是JAR文件的路径及作业的输入和输出路径。在**JAR Location**区域，输入已上传的JAR文件的位置。假如JAR文件命名为`wc1.jar`，并且上传到了名为`mybucket`的桶中，那么路径应为`mybucket/wc1.jar`.

在**JAR Arguments**区域，需要键入主类的名称及作业的输入和输出位置。对于**S3**上的文件，我们可以使用形如`s3://bucketname/objectname`的URL。点击**Continue**按钮，为作业流选用虚拟机的熟悉界面出现了，如下图所示。



现在，继续完成作业的安装和执行，就如我们在第2章所做的那样。

原理分析

在EMR上，我们可以重用为本地Hadoop集群所写的代码，这点值得特别注意。此外，除了前面这几个步骤，不管要执行的作业代码的源文件是什么，EMR控制台的大部分设置都是一样的。

在本章剩余部分，我们不会明确展示正在EMR上执行的代码，而是将更多地关注本地集群，因为在EMR上运行JAR文件是很容易的。

3.8.1　0.20 之前版本的Java MapReduce API

本书优先使用0.20及以上版本的MapReduce Java API，但是基于下面两个原因，我们需要快

速浏览一下旧API。

(1) 许多在线示例和其他参考材料是用旧API编写的。
(2) MapReduce框架的若干内容还没有移植到新API，我们不得不使用旧API来研究这些内容。

旧版API类主要在org.apache.hadoop.mapred包中。

新版API类使用具体的Mapper和Reducer类，而旧版API倾向于使用抽象类和接口。

Mapper类的实现继承了MapReduceBase抽象类并实现了Mapper接口，而一个自定义的Reducer类将继承相同的MapReduceBase抽象类，但实现了Reducer接口。

由于MapReduceBase的功能是处理作业设置与配置，它不是理解MapReduce模型的核心内容，所以我们不会讨论其太多的细节。但0.20之前版本的Mapper和Reducer的接口是值得一看的。

```
public interface Mapper<K1, V1, K2, V2>
{
void map( K1 key, V1 value, OutputCollector< K2, V2> output, Reporter reporter) throws
IOException ;
}

public interface Reducer<K2, V2, K3, V3>
{
void reduce( K2 key, Iterator<V2> values,
OutputCollector<K3, V3> output, Reporter reporter)
throws IOException ;
}
```

这里需要理解下列几点内容。

- OutputCollector类的泛型参数更明确地展示了上述方法的结果是如何输出的。
- 旧版API为此使用OutputCollector类，并且使用Reporter类将状态和指标信息写入Hadoop框架。0.20版的API将这些功能整合到了Context类。
- Reducer接口使用Iterator对象代替Iterable对象。这个变化产生的原因是后者更符合Java语法而且使得代码更简洁。
- 在旧版API中，无论是map方法还是reduce方法都能抛出InterruptedException异常。

如你所见，不同版本的API发生的变化改变了MapReduce程序的编写方式，但不会改变mapper或reducer的用途或功能。除非需要，不要觉得你必须成为这两套API的专家。熟悉任意一套API都能跟上本书剩余部分的内容。

3.8.2 Hadoop提供的mapper和reducer实现

我们并非总是必须从头开始编写自己的Mapper和Reducer类。Hadoop提供了几种常见的Mapper和Reducer的实现，它们可以用于我们的作业当中。如果我们不重写新版API的Mapper

和Reducer类的任何方法，默认的实现是恒等的Mapper和Reducer类，即仅仅将输入原封不动地输出。

需要注意的是，随着时间的推移，可能会有更多的预先写好的Mapper和Reducer类加入到Hadoop API中。目前，新版API中预先写好的Mapper和Reducer类的数量不如旧版API多。

mapper可以在org.apache.hadoop.mapreduce.lib.mapper找到，并包含以下内容。

- InverseMapper：该类输出（value，key）。
- TokenCounterMapper：它对输入的每行文本的离散令牌进行计数。

reducer可以在org.apache.hadoop.mapreduce.lib.reduce找到，目前包含以下内容。

- IntSumReducer：它输出每个键对应的整数值列表的总和。
- LongSumReducer：它输出每个键对应的长整型值列表的总和。

3.9　实践环节：WordCount 的简易方法

让我们重温WordCount，但这次使用一些预定义的map和reduce类：

(1) 新建一个包含下列代码的WordCountPredefined.java文件：

```
import org.apache.hadoop.conf.Configuration ;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.map.TokenCounterMapper;
import org.apache.hadoop.mapreduce.lib.reduce.IntSumReducer;

 public class WordCountPredefined
{
     public static void main(String[] args) throws Exception
     {
         Configuration conf = new Configuration();
         Job job = new Job(conf, "word count1");
         job.setJarByClass(WordCountPredefined.class);
         job.setMapperClass(TokenCounterMapper.class);
         job.setReducerClass(IntSumReducer.class);
         job.setOutputKeyClass(Text.class);
         job.setOutputValueClass(IntWritable.class);
         FileInputFormat.addInputPath(job, new Path(args[0]));
         FileOutputFormat.setOutputPath(job, new Path(args[1]));
         System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

(2) 现在，编译上述文件，创建JAR文件，并和之前一样运行它。

(3) 如果想要使用相同的输出位置，别忘了要在运行作业之前删除输出目录。例如，使用 `hadoop fs -rmr output`。

原理分析

由于MapReduce普遍使用WordCount作为入门示例，即便使用预定义的`Mapper`和`Reducer`实现整个WordCount方案，可能也不会令人惊讶。`TokenCounterMapper`类仅仅将每行输入切分为一系列的`(token,1)`对，`IntSumReducer`类通过累加每个键对应值的数量提供一个最终计数。

这里要注意两点重要内容。

- 虽然对预定义的Mapper和Reducer来讲，使用它们实现WordCount无疑是鼓舞人心的，但它们并非专用于WordCount程序，而是可以被广泛使用的。
- 复用现有的mapper和reducer是需要牢记的经验。尤其是，通常实现一个新MapReduce作业的最好起点，往往是改动现有的作业。

3.10 查看 WordCount 的运行全貌

为了更详细地探讨mapper和reducer之间的关系，并揭示Hadoop的一些内部工作机理，现在我们将全景呈现WordCount（或任意MapReduce作业）是如何执行的。

3.10.1 启动

调用驱动中的`Job.waitForCompletion()`是所有行动的开始。该驱动程序是唯一一段运行在本地机器上的代码，上述调用开启了本地主机与JobTracker的通信。请记住，JobTracker负责作业调度和执行的各个方面，所以当执行任何与作业管理相关的任务时，它成为了我们的主要接口。JobTracker代表我们与NameNode通信，并对储存在HDFS上的数据相关的所有交互进行管理。

3.10.2 将输入分块

这些交互首先发生在JobTracker接受输入数据，并确定如何将其分配给map任务的时候。回想一下，HDFS文件通常被分成至少64 MB的数据块，JobTracker会将每个数据块分配给一个map任务。

当然，WordCount示例涉及的数据量是微不足道的，它刚好适合放在一个数据块中。设想一个更大的以TB为单位的输入文件，切分模型变得更有意义。每段文件（或用MapReduce术语来讲，每个**split**）由一个map作业处理。

一旦对各分块完成了运算，JobTracker就会将它们和包含Mapper与Reducer类的JAR文件放置在HDFS上作业专用的目录，而该路径在任务开始时将被传递给每个任务。

3.10.3　任务分配

一旦JobTracker确定了所需的map任务数，它就会检查集群中的主机数，正在运行的TaskTracker数以及可并发执行的map任务数（用户自定义的配置变量）。JobTracker也会查看各个输入数据块在集群中的分布位置，并尝试定义一个执行计划，使TaskTracker尽可能处理位于相同物理主机上的数据块。或者即使做不到这一点，TaskTracker至少处理一个位于相同硬件机架中的数据块。

数据局部性优化是Hadoop能高效处理巨大数据集的一个关键原因。还记得，默认情况下，每个数据块会被复制到三台不同的主机。所以，在本地处理大部分数据块的任务/主机计划比起初预想的可能性更高。

3.10.4　任务启动

然后，每个TaskTracker开启一个独立的Java虚拟机来执行任务。这确实增加了启动时间损失，但它将因错误运行map或reduce任务所引发的问题与TaskTracker隔离开来，而且可以将它配置成在随后执行的任务之间共享。

如果集群有足够的能力一次性执行所有的map任务，它们将会被全部启动，并获得它们将要处理的分块数据和作业JAR文件。每个TaskTracker随后将分块复制到本地文件系统。

如果任务数超过了集群能力，JobTracker将维护一个挂起任务队列，并在节点完成最初分配的map任务后，将挂起任务分配给节点。

现在，我们准备查看map任务执行完毕的数据。听起来工作量似乎很大，事实确实如此。这也解释了在运行任意MapReduce作业时，为什么系统启动及执行上述步骤会花费大量时间。

3.10.5　不断监视JobTracker

现在，JobTracker等待TaskTracker执行所有的mapper和reducer。它不断地与TaskTracker交换心跳和状态消息，查找进度或问题的证据。它还从整个作业执行过程的所有任务中收集指标，其中一些指标是Hadoop提供的，还有一些是map和reduce任务的开发人员指定的，不过本例中我们没有使用任何指标。

3.10.6 mapper的输入

在第2章中，WordCount的输入是一个仅有一行内容的文本文件。在本节剩余部分，假设输入文件是一个极为普通的两行文本文件。

```
This is a test
Yes this is
```

驱动类使用TextInputFormat指定了输入文件的格式和结构，因此，Hadoop会把输入文件看做以行号为键并以该行内容为值的文本。因此mapper的两次调用将被赋予以下输入。

```
1 This is a test
2 Yes it is.
```


3.10.7 mapper的执行

根据作业配置的方式，mapper接收到的键/值对分别是相应行在文件中的偏移量以及该行内容。因为我们不关心每行文本在文件中的位置，所以`WordCountMapper`类的map方法舍弃了键，并使用标准的Java String类的`split`方法将每行文本内容拆分成词。需要注意的是，使用正则表达式或`StringTokenizer`类可以更好地断词，但对于我们的需求，这种简单的方法就足够了。

然后，针对每个单独的词，mapper输出由单词本身组成的键和值1。



我们加入了一些优化措施，但现在不必对其考虑太多。读者会看到，我们并不是每次创建包含值1的`IntWritable`对象，而是以静态变量的形式创建`IntWritable`对象，并在每次调用时复用该对象。类似地，我们使用一个`Text`对象并在每次执行函数时重置其内容。这样做的原因是，尽管它对我们小型的输入文件帮助不大，但处理巨大数据集时，可能会对mapper进行成千上万次调用。如果每次调用都为输出的键和值创建一个新对象，这将消耗大量的资源，同时垃圾回收会引发更频繁的停滞。我们使用这个值，知道`Context.write`方法不会对其进行改动。

3.10.8 mapper的输出和reducer的输入

mapper的输出是一系列形式为`(word,1)`的键值对。本例中，mapper的输出为：

```
(This,1), (is, 1), (a, 1), (test., 1), (Yes, 1), (it, 1), (is, 1)
```

这些从mapper输出的键值对并不会直接传给reducer。在map和reduce之间，还有一个shuffle阶段，这也是许多MapReduce奇迹发生的地方。

3.10.9　分块

Reduce接口的隐性保证之一是，与给定键相关的所有值都会被提交到同一个reducer。由于一个集群中运行着多个reduce任务，因此，每个mapper的输出必须被分块，使其分别传入相应的各个reducer。这些分块文件保存在本地节点的文件系统。

集群中的Reduce任务数并不像mapper数量一样是动态的，事实上，我们可以在作业提交阶段指定reduce任务数。因此，每个TaskTracker就知道集群中有多少个reducer，并据此得知mapper输出应切分为多少块。



在后续章节中，我们会介绍故障容忍，但现在明显的问题是，假如reducer失败了，会给本次计算带来什么影响呢？答案是，JobTracker会保证重新执行发生故障的reduce任务，可能是在不同的节点重新执行，因此临时故障不是问题。更为严重的问题是，数据块中的数据敏感性缺陷或错误数据可能导致整个作业失败，除非采取一些手段。

3.10.10　可选分块函数

Partitioner类在org.apache.hadoop.mapreduce包中，该抽象类具有如下特征：

```
public abstract class Partitioner<Key, Value>
{
public abstract int getPartition( Key key, Value value,
int numPartitions) ;
}
```

默认情况下，Hadoop将对输出的键进行哈希运算，从而实现分块。此功能由org. apache. hadoop.mapreduce.lib.partition包里的HashPartitioner类实现，但某些情况下，用户有必要提供一个自定义的Partitioner子类，在该子类中实现针对具体应用的分块逻辑。特别是当应用标准哈希函数导致数据分布极不均匀时，自定义Partitioner子类尤为必要。

3.10.11　reducer类的输入

reducer的TaskTracker从JobTracker接收更新，这些更新指明了集群中哪些节点承载着map的输出分块，这些分块将由本地reduce任务处理。之后，TaskTracker从各个节点获取分块，并将它们合并为一个文件反馈给reduce任务。

3.10.12 reducer类的执行

我们实现的WordCountReducer类很简单。针对每个词，该类仅对数组中的元素数目进行统计并为每个词输出最终的(Word, count)键值对。



不必为避免创建多余对象而过多地考虑采取一些优化措施。reducer的调用次数通常小于mapper的调用次数，因此调用reducer带来的开销无需特别在意。然而，假如读者有着非常严格的性能要求，那么您可以采取任何有利于提高性能的措施。

我们调用WordCount处理示例输入，除is出现两次外，其余所有词仅出现一次。



请注意,this和This分别进行了计数，这是因为我们并不打算忽略大小写。类似地，每个句子以句号结尾，这个规则可能干扰is的值，因为is与is..并不相同。在处理文本数据时，对大写、标点符号、连字符、页码及其他方面都要特别小心，因为这些东西可以误导数据的理解方式。在这些情况下，通常需要一个先期的MapReduce作业对数据集执行标准化或清理策略。

3.10.13 reducer类的输出

因此，本例中reducer的最终输出集合为：

```
(This, 1), (is, 2), (a, 1), (test, 1), (Yes, 1), (this, 1)
```

这些数据将被输出到驱动程序指定的输出路径下的分块文件中，并将使用指定的OutputFormat对其进行格式化。每个reduce任务写入一个以part-r-nnnnn为文件名的文件，其中nnnnn从00000开始并逐步递增。当然，这些内容曾在第2章讲到，希望part前缀现在稍微容易理解一些。

3.10.14 关机

一旦成功完成所有任务，JobTracker向客户端输出作业的最终状态，以及作业运行过程中一些比较重要的计数器集合。完整的作业和任务历史记录存储在每个节点的日志路径中，通过JobTracker的网络用户接口更易于访问，只需将浏览器指向JobTracker节点的50030端口即可。

3.10.15　这就是MapReduce的全部

如你所见，Hadoop为每个MapReduce程序提供了大量机制，同时，Hadoop提供的框架在许多方面都进行了简化。如前所述，对于WordCount这样的小程序来说，MapReduce的大部分机制并没有多大价值，但是不要忘了，无论在本地Hadoop或EMR，我们可以使用相同的软件和mapper/reducer在巨大的集群上对更大的数据集进行字数统计。那时，Hadoop所做的大量工作使用户能够在如此大的数据集上进行数据分析。否则，手工实现代码分发、代码同步以及并行运算将付出超乎想象的努力。

3.10.16　也许缺了combiner

前面讲述了MapReduce程序运行的各个步骤，却漏掉了另外一个可选步骤。在reducer获取map方法的输出之前，Hadoop允许使用combiner类对map方法的输出执行一些前期的排序操作。

为什么要有combiner

Hadoop设计的前提是，减少作业中成本较高的部分，通常指磁盘和网络输入输出。mapper的输出往往是巨大的——它的大小通常是原始输入数据的许多倍。Hadoop的一些配置选项可以帮助减少reducer在网络上传输如此大的数据量所带来的性能影响。combiner则采取了不同的方法，它对数据进行早期聚合以减少所需传输的数据量。

combiner没有自己的接口，它必须具有与reducer相同的特征，因此也要继承org.apache.hadoop.mapreduce包里的Reduce类。这样做的效果主要是，在map节点上对发往各个reducer的输出执行mini-reduce操作。

Hadoop不保证combiner是否被执行。有时候，它可能根本不执行，而某些时候，它可能被执行一次、两次甚至多次，这取决于mapper为每个reducer生成的输出文件的大小和数量。

3.11　实践环节：使用 combiner 编写 WordCount

让我们在第一个WordCount示例程序中加入combiner。事实上，我们可以使用reducer作为combiner。因为combiner必须具有和reducer相同的接口，所以经常会看到使用reducer作为combiner的情况，但要注意，包含在reducer中的处理类型将确定其是否可用作combiner，稍后我们将讨论这个问题。由于我们试图统计单词出现的次数，可以在map节点进行部分计数，并将这些结果传给reducer。

(1) 复制WordCount1.java，保存为WordCount2.java，并在Mapper和Reducer类定义之间加入下列内容来修改驱动类。

```
job.setCombinerClass(WordCountReducer.class);
```

(2) 修改类名为WordCount2并编译。

```
$ javac WordCount2.java
```

(3) 创建JAR文件。

```
$ jar cvf wc2.jar WordCount2*class
```

(4) 在Hadoop上运行作业。

```
$ hadoop jar wc2.jar WordCount2 test.txt output
```

(5) 检查输出。

```
$ hadoop fs -cat output/part-r-00000
```


原理分析

本次输出可能与预期有所差别，因为is出现了两次，其值却被错误统计为1。

问题在于combiner与reducer的交互方式。提交给reducer的值，之前是(is, 1, 1)，现在却是(is, 2)，这是因为combiner对每个词的出现次数自行进行合并。然而，reducer并没有看到Iterable对象中的真正值，它仅对combiner提交的数据进行了统计。

使用reducer作为combiner的条件

编写combiner时要特别留心。请记住，Hadoop不保证combiner被应用到map输出的次数，可能根本不执行，也可能执行1次或者多次。因此，关键问题在于，由combiner执行的操作是否可被如此应用。分布式的操作，如总和、加法或类似操作通常是安全的，但是，如前所示，务必确保reduce逻辑不包含可能破坏这个特性的隐含假设。

3.12　实践环节：更正使用 combiner 的 WordCount

让我们对WordCount做一些必要的修改，以正确使用combiner。

将WordCount2.java复制为名为WordCount3.java的新文件，并将reduce方法改为如下内容：

```
public void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException
{
int total = 0 ;
for (IntWritable val : values))
{
```

```
total+= val.get() ;
}
            context.write(key, new IntWritable(total));
}
```

别忘了，要将该类名改为WordCount3，然后编译它，创建JAR文件，并和以前一样运行作业。

原理分析

现在，输出与预期一致。map端对combiner的调用全部执行成功，同时，reducer生成的输出值全都正确。



假如原来的reducer被用作combiner而新的reducer被用作reducer，WordCount程序会工作正常吗？答案是否定的，但是我们的测试样例无法说明这一点。由于combiner可能会在map输出数据上调用多次，如果数据集足够大的话，同样的错误会多次出现在map输出中，但是由于此处输入数据规模小，错误没有显现出来。从根本上讲，原来的reducer是错误的，但其错误并不明显，要小心此类细微的逻辑缺陷。事实上，这类问题难以调试，因为代码在开发机上的数据集的子集工作正常，却在更大的业务集群上发生故障。要仔细推敲combiner类，而不能完全依赖于处理小样本数据的测试。

复用助您一臂之力

在前面章节中，我们使用了现有的作业类文件并对其进行了改动。这是一个非常常见的Hadoop开发流程的小例子——使用现有的作业文件作为开发新作业的起点。虽然新作业的mapper和reducer逻辑与现有作业相去甚远，但使用现有作业通常能够节省时间，因为它能够帮您记住实现mapper、reducer和驱动所必须的元素。

随堂测验：MapReduce的结构

问题1　你必须为MapReduce作业指定哪些内容？

(1) mapper和reducer类。

(2) mapper、reducer和combiner类。

(3) mapper、reducer、partitioner和combiner类。

(4) 什么都不需要，所有类都有默认实现。

问题2　Combiner会被执行多少次？

(1) 至少1次。

(2) 0次或1次。

(3) 0次、1次或多次。

(4) 这是可配置的。

问题3 有一个mapper为每个键生成一个整数值，下面是reduce操作。

- Reducer A：输出整数值集合的总和。
- Reducer B：输出值集合中的最大值。
- Reducer C：输出值集合的平均值。
- Reducer D：输出集合中最大值与最小值之差。

这些reduce操作中，哪个可被安全地用作combiner?

(1) 全部。

(2) A和B。

(3) A、B和D。

(4) C和D。

(5) 都不可以。

3.13 Hadoop 专有数据类型

截至目前，我们已经草草了解了用作map和reduce类的输入和输出的数据类型。现在，让我们研究一下这些数据类型。

3.13.1 Writable和WritableComparable接口

如果浏览`org.apache.hadoop.io`包的Hadoop API，读者会发现一些熟悉的类带有`Writable`词缀，这些类包括`Text`、`IntWritable`及其他。

`org.apache.hadoop.io`也包含了`Writable`接口的定义，其定义如下：

```
import java.io.DataInput ;
import java.io.DataOutput ;
import java.io.IOException ;

public interface Writable
{
void write(DataOutput out) throws IOException ;
void readFields(DataInput in) throws IOException ;
}
```

Writable接口的主要目的是，当数据在网络上传输或从硬盘读写时，提供数据的序列化和反序列化机制。所有用作mapper或reducer输入或输出值的数据类型（也就是说，`V1`、`V2`或`V3`）都必须实现这个接口。

用作键的数据(K1，K2，K3)有着更为严格的要求：除Writable之外，它必须实现标准Java中的Comparable接口：

```
public interface Comparable
{
public int compareTO( Object obj) ;
}
```

compare方法的返回值为-1、0或者1，这取决于被比较的对象小于、等于或大于当前对象。

作为一个便用接口，Hadoop在org.apache.hadoop.io包里提供了一个WritableComparable接口。

```
public interface WritableComparable extends Writable, Comparable
{}
```

3.13.2　wrapper类介绍

幸运的是，你不必从头开始。正如你所看到的，Hadoop提供了包装Java原始类型并实现WritableComparable的类。它们被放置在org.apache.hadoop.io包。

1. 原始包装类

这些类在概念上与原始包装类相似，如java.lang中的Integer和Long。它们保持一个原始值，该值既可以在创建类的时候设置，也可以通过setter方法设置。

- BooleanWritable
- ByteWritable
- DoubleWritable
- FloatWritable
- IntWritable
- LongWritable
- VIntWritable：可变长度的整数类型
- VLongWritable：可变长度的长整数类型

2. 数组包装类

这些类为其他Writable对象数组提供了可写封装。例如，这些类的实例可以存储IntWritable或DoubleWritable类型的数组，却不能存储原始的整数或浮点数类型的数组。这些类需要继承Writable类。如下所示：

- ArrayWritable
- TwoDArrayWritable

3. Map包装类

这些类允许使用java.util.Map接口作为键或者值。需要注意的是，它们被定义为Map<Writable, Writable>，并有效管理部分内部运行时类型检查。这就意味着弱化了编译类型检查，所以要当心。

- AbstractMapWritable：这是其他具体的Writable map包装类的基类。
- MapWritable：这是一个通用的map包装类，将Writable键映射为Writable值。
- SortedMapWritable：这是MapWritable类的一个特殊实现，它同时也实现了SortedMap接口。

3.14 实践环节：使用 Writable 包装类

让我们编写一个类，来展示实际使用中的包装类。

(1) 创建WritablesTest.java文件，键入下列内容：

```
import org.apache.hadoop.io.* ;
import java.util.* ;

public class WritablesTest
{
   public static class IntArrayWritable extends ArrayWritable
   {
      public IntArrayWritable()
      {
         super(IntWritable.class) ;
      }
   }

   public static void main(String[] args)
   {
System.out.println("*** Primitive Writables ***") ;
        BooleanWritable bool1 = new BooleanWritable(true) ;
        ByteWritable byte1 = new ByteWritable( (byte)3) ;
        System.out.printf("Boolean:%s Byte:%d\n", bool1, byte1.
get()) ;

        IntWritable i1 = new IntWritable(5) ;
        IntWritable i2 = new IntWritable( 17) ;    System.
out.printf("I1:%d I2:%d\n", i1.get(), i2.get()) ;
        i1.set(i2.get()) ;
        System.out.printf("I1:%d I2:%d\n", i1.get(), i2.get()) ;
        Integer i3 = new Integer( 23) ;
        i1.set( i3) ;
        System.out.printf("I1:%d I2:%d\n", i1.get(), i2.get()) ;

System.out.println("*** Array Writables ***") ;
        ArrayWritable a = new ArrayWritable( IntWritable.class) ;
```

```
        a.set( new IntWritable[]{ new IntWritable(1), new
IntWritable(3), new IntWritable(5)}) ;

        IntWritable[] values = (IntWritable[])a.get() ;

        for (IntWritable i: values)


        System.out.println(i) ;

        IntArrayWritable ia = new IntArrayWritable() ;
        ia.set( new IntWritable[]{ new IntWritable(1), new
IntWritable(3), new IntWritable(5)}) ;

        IntWritable[] ivalues = (IntWritable[])ia.get() ;

        ia.set(new LongWritable[]{new LongWritable(1000l)}) ;

System.out.println("*** Map Writables ***") ;
        MapWritable m = new MapWritable() ;
        IntWritable key1 = new IntWritable(5) ;
        NullWritable value1 = NullWritable.get() ;
        m.put(key1, value1) ;
        System.out.println(m.containsKey(key1)) ;
        System.out.println(m.get(key1)) ;
        m.put(new LongWritable(1000000000), key1) ;
        Set<Writable> keys = m.keySet() ;

        for(Writable w: keys)
        System.out.println(w.getClass()) ;
    }
}
```

(2) 编译并运行该类，你会得到下列输出：

```
*** Primitive Writables ***
Boolean:true Byte:3
I1:5 I2:17
I1:17 I2:17
I1:23 I2:17
*** Array Writables ***
1
3
5
*** Map Writables ***
true
(null)
class org.apache.hadoop.io.LongWritable
class org.apache.hadoop.io.IntWritable
```

原理分析

上述输出的含义在很大程度上是不言自明的。我们创建了多个Writable包装对象，并展示

了它们的一般用法。其中，有几个关键点。

- 如前所述，Writable 保证了自身类型安全性。因此，可能有存储多种数据类型的数组或map，如上述示例代码所示。
- 我们可以使用自动拆箱，例如，将一个Integer对象提供给参数为IntWritable的方法，该方法希望收到的参数是一个int变量。
- 如果ArrayWritable被用作reduce函数的输入，还需要一个内部类，必须定义一个带有默认构造函数的子类。

1. 其他包装类

- CompressedWritable：这是一个基类，它允许大型对象保持压缩，直到其属性被显式访问。
- ObjectWritable：这是一个多用途的通用对象包装类。
- NullWritable：这是一个表示空值的对象。
- VersionedWritable：这是一个允许Writable类跟踪版本号的基本实现。

一展身手：练习Writables类

练习使用NullWritable和ObjectWritable编写类，如同之前的例子一样。

2. 编写自定义类

正如读者在Writable和Comparable接口所看到的，所需方法相当简单；如果读者想在MapReduce作业中使用自定义类作为键或者值，不必害怕增加这样的功能。

3.15 输入/输出

我们已多次提到驱动类的一个功能，却没有进行详细解释：指定MapReduce作业的输入和输出的数据格式和结构。

3.15.1 文件、split和记录

我们已经讨论过，在作业启动时，文件被切分为split，split中的数据被送往mapper。然而，却忽视了两个方面：数据在文件中如何存储以及单个键值如何传给mapper。

3.15.2 InputFormat和RecordReader

对于第一个问题，Hadoop提出了InputFormat的概念。org.apache.hadoop.mapreduce

包里的InputFormat抽象类提供了如下列代码所示的两个方法。

```
public abstract class InputFormat<K, V>
{
public abstract List<InputSplit> getSplits( JobContext context) ;
RecordReader<K, V> createRecordReader(InputSplit split,
TaskAttemptContext context) ;
}
```

这些方法展示了InputFormat类的两个功能：

- 将输入文件切分为map处理所需要的split；
- 创建RecordReader类，它将从一个split生成键值对序列。

RecordReader类同样也是org.apache.hadoop.mapreduce包里的抽象类。

```
public abstract class RecordReader<Key, Value> implements Closeable
{
public abstract void initialize(InputSplit split, TaskAttemptContext
context) ;
   public abstract boolean nextKeyValue()
throws IOException, InterruptedException ;
public abstract Key getCurrentKey()
throws IOException, InterruptedException ;
public abstract Value getCurrentValue()
throws IOException, InterruptedException ;
public abstract float getProgress()
throws IOException, InterruptedException ;
public abstract close() throws IOException ;
}
```

为每个split创建一个RecordReader实例，该实例调用getNextKeyValue并返回一个布尔值，该值指明下一个键值对是否可用。如果答案是肯定的，getKey和getValue方法被用于分别访问键和值。

因此，组合使用InputFormat和RecordReader可以将任何类型的输入数据转换为MapReduce所需的键值对。

3.15.3 Hadoop提供的InputFormat

Hadoop在org.apache.hadoop.mapreduce.lib.input包里提供了一些InputFormat的实现。

- FileInputFormat：这是一个抽象基类，它可以作为任何基于文件输入的父类。
- SequenceFileInputFormat：这是一个高效的二进制文件格式，我们将在下一节讨论它。
- TextInputFormat：它用于普通文本文件。



0.20之前版本的API在org.apache.hadoop.mapred包里定义了一些其他的InputFormat。

请注意，InputFormat并不局限于从文件读取数据，FileInputFormat本身就是InputFormat的子类。Hadoop有可能使用不基于文件的数据作为MapReduce作业的输入，常见的数据源是关系数据库或HBase。

3.15.4 Hadoop提供的RecordReader

类似地，Hadoop在org.apache.hadoop.mapreduce.lib.input包里也提供了一些常见的RecordReader实现。

- LineRecordReader：这是RecordReader类对文本文件的默认实现，它将行号视为键并将该行内容视为值。
- SequenceFileRecordReader：该类从二进制文件SequenceFile读取键/值

同样，0.20版本之前的API在org.apache.hadoop.mapred包里提供了其他RecordReader类，例如KeyValueRecordReader，它们没有被迁移到新API。

3.15.5 OutputFormat和RecordWriter

采用类似模式，org.apache.hadoop.mapreduce包里的OutputFormat和RecordWriter的子类负责共同写入作业输出。本节内容不讨论它们的细节，但总体方法是相似的，尽管OutputFormat为验证输出规范实现了更为复杂的API。



如果指定的输出路径已经存在，该步骤将引发作业失败。如果你想改变这种情况，需要一个重写该方法的OutputFormat子类。

3.15.6 Hadoop提供的OutputFormat

org.apache.hadoop.mapreduce.output包提供了下列OutputFormat类。

- FileOutputFormat：这是所有基于文件的OutputFormat的基类。
- NullOutputFormat：这是一个虚拟类，它丢弃所有输出并对文件不做任何写入。
- SequenceFileOutputFormat：它将输出写入二进制SequenceFile。
- TextOutputFormat：它把输出写入到普通文本文件。

请注意，上述类把它们所需的RecordWriter定义为内部类，因此，不存在单独实现的RecordWriter类。

3.15.7　别忘了Sequence files

org.apache.hadoop.io包里的SequenceFile类提供了高效的二进制文件格式，它经常用于MapReduce作业的输出。尤其是当作业的输出被当做另一个作业的输入时。Sequence文件有如下几个优点：

- 作为二进制文件，它们本质上比文本文件更为紧凑；
- 它们支持不同层面的可选压缩，也就是说，可以对每条记录或整个split进行压缩；
- 该文件可被并行切分和处理。

最后一个特性很重要，大多数二进制格式（尤其是压缩或加密文件）是无法切分的，必须以单独的线性数据流的形式读取。使用这种无法切分的文件作为MapReduce作业的输入，意味着需要使用一个单独的mapper处理整个文件，造成潜在的巨大性能损失。在此情况下，最好使用可切分的格式，如SequenceFile，或者在无法避免接收其他格式文件的情况下，执行预处理步骤将其转换成可切分的格式。这需要权衡利弊，因为文件格式转换也需要一定的时间。但在很多情况下，尤其是处理复杂的map任务时，使用可切分格式所节省的时间将超过文件格式转换的时间。

3.16　小结

本章讲述了MapReduce的大量基础知识，现在，我们具备了深入研究MapReduce的基础。特别是，我们学习了键值对是被广泛应用的数据模型，它可以很好地应用于MapReduce处理。我们也学习了如何使用0.20及以上版本的Java API编写mapper和reducer。

随后，我们了解了MapReduce作业是如何工作的，以及map与reduce方法是怎样通过大量的协作和任务调度机制紧密结合在一起的。我们还了解到，某些MapReduce作业需要自定义的partitioner或combiner。

我们还研究了Hadoop如何从文件系统读取数据并将输出数据写入文件系统。它使用InputFormat和OutputFormat的概念将文件作为整体进行处理，并使用RecordReader将数据格式转化为键值对，然后使用RecordWriter将数据格式从键值对转换为所需格式。

有了这些知识，我们将在下一章转到案例学习。该案例展示了如何持续开发和改进处理大数据集的MapReduce程序。

第4章

开发MapReduce程序

既然我们已经探索了MapReduce技术，本章将介绍如何使用MapReduce解决实际问题。特别是，我们将以更大规模的数据集为例，探索使用MapReduce提供的工具分析数据集的方法。

本章包括以下内容：

- Hadoop Streaming及其使用；
- UFO目击事件数据集；
- 使用Streaming作为开发或调试工具；
- 在一个作业中使用多个mapper；
- 在集群上高效共享实用程序文件和数据；
- 报告作业和任务的状态信息及可用于调试的日志信息。

本章不仅要介绍具体的工具，同时也要介绍如何分析新数据集。我们将从探寻如何使用脚本编程语言辅助MapReduce进行原型设计和初步分析入手。上一章还在讲Java API，本章却马上换了另一种语言，这似乎看起来很奇怪，但我们的目的是让读者意识到，可以使用不同方法解决所面临的问题。虽然很多作业没必要用Java API以外的其他语言来实现，但在某些情况下，使用其他方法才是最合适的。相信这些技术为你增加了新的备选工具，并且有了这些经验，你会更容易知道在给定场景中，哪种方法才是最佳解决方案。

4.1 使用非 Java 语言操作 Hadoop

我们前面提到过，并非必须使用Java语言才能编写MapReduce程序。虽然大多数程序是用Java编写的，但由于某些原因，读者希望或需要用另一种语言编写map和reduce任务。比如，读者有一些现有代码可供利用，或者需要使用第三方的二进制文件等，原因不一而足却又合情合理。

Hadoop提供了一些辅助非Java开发的机制，其中主要包括**Hadoop Pipes**和**Hadoop Streaming**。前者提供了Hadoop的原生C++接口，后者则允许将任何使用标准输入和输出的程序用于map和reduce任务。本章，我们将大量使用Hadoop Streaming。

4.1.1 Hadoop Streaming工作原理

map和reduce任务利用MapReduce Java API提供实现任务功能的方法。这些方法将其参数视为任务输入，并通过`Context`对象输出结果。这是一个明确的并且类型安全的接口，但是专为Java定义的。

Hadoop Streaming采用了不同的方法。使用Streaming编写的map任务每次从标准输入读取一行内容作为任务输入，并将任务结果输出到标准输出。reduce任务执行同样的操作，而且其数据流同样使用标准输入和输出。

任何可以读写标准输入和输出的程序都可以用于Streaming，例如已编译的二进制文件、Unixshell脚本，或用像Ruby或Python这样的动态语言编写的程序等。

4.1.2 使用Hadoop Streaming的原因

Streaming最大的优势在于，使用它可以比使用Java更快地反复尝试不同的想法。与“编译/jar/提交”的Java开发周期不同，用户只需编写脚本，并把它们作为参数传给Streaming jar文件。尤其是当对新数据集进行初步分析或尝试新想法的时候，Streaming可以明显加快开发进度。

动态语言和静态语言各有所长，动态语言适合敏捷开发，而静态语言具有运行性能和类型检查的优势。换句话说，静态语言有的，正是动态语言没有的，反之亦然。当使用Streaming时，动态语言的缺点同样存在。因此，我们倾向于在前期分析时使用Streaming，而在实现运行于产品集群的作业时使用Java语言。

本章中，我们将在Streaming例子中使用Ruby语言，但这只是个人喜好。如果读者喜欢用shell脚本或其他语言，比如Python，那么可以把Ruby脚本转换成自己选择的语言。

4.2 实践环节：使用 Streaming 实现 WordCount

让我们老调重弹，通过执行下列步骤使用Streaming再次实现WordCount。

(1) 将以下内容保存为`wcmapper.rb`文件。

```
#/bin/env ruby

while line = gets
```

```
    words = line.split("\t")
    words.each{ |word| puts word.strip+"\t1"}}
end
```

(2) 通过执行下列命令使wcmapper.rb成为可执行文件。

```
$ chmod +x wcmapper.rb
```

(3) 将以下文件保存为 wcreducer.rb。

```
#!/usr/bin/env ruby

current = nil
count = 0

while line = gets
    word, counter = line.split("\t")

    if word == current
        count = count+1
    else
        puts current+"\t"+count.to_s if current
        current = word
        count = 1
    end
end
puts current+"\t"+count.to_s
```

(4) 通过执行下列命令使wcreducer.rb成为可执行文件。

```
$ chmod +x wcreducer.rb
```

(5) 利用上一章的数据文件，以Streaming作业的形式执行脚本。

```
$ hadoop jar hadoop/contrib/streaming/hadoop-streaming-1.0.3.jar
-file wcmapper.rb -mapper wcmapper.rb -file wcreducer.rb
-reducer wcreducer.rb -input test.txt -output output
packageJobJar: [wcmapper.rb, wcreducer.rb, /tmp/hadoop-
hadoop/hadoop-unjar1531650352198893161/] [] /tmp/
streamjob937274081293220534.jar tmpDir=null
12/02/05 12:43:53 INFO mapred.FileInputFormat: Total input paths
to process : 1
12/02/05 12:43:53 INFO streaming.StreamJob: getLocalDirs(): [/var/
hadoop/mapred/local]
12/02/05 12:43:53 INFO streaming.StreamJob: Running job:
job_201202051234_0005
...
12/02/05 12:44:01 INFO streaming.StreamJob:    map 100%    reduce 0%
12/02/05 12:44:13 INFO streaming.StreamJob:    map 100%    reduce 100%
12/02/05 12:44:16 INFO streaming.StreamJob: Job complete:
job_201202051234_0005
12/02/05 12:44:16 INFO streaming.StreamJob: Output: wcoutput
```

(6) 检查结果文件。

```
$ hadoop fs -cat output/part-00000
```

原理分析

就这个例子而言，实际上不用理会Ruby代码的细节，即使读者不懂这门语言也不要紧。

首先，我们创建了一个脚本，它将作为我们的mapper。该脚本使用`gets`函数从标准输入中读取一行，将该行切分为词语，并使用`puts`函数将词与值`1`写到标准输出。然后，我们把该脚本做成可执行文件。

本例中，reducer则稍微复杂一些，其原因将在下一节中描述。无论如何，它按照我们的期望执行了作业，对每个单词出现的次数进行计数，从标准输入读取数据，并将输出的最终值提交到标准输出。同样，我们把该脚本做成可执行文件。

需要注意的是，在这两种情况下，我们隐式地使用了前几章讨论的Hadoop输入和输出格式。`TextInputFormat`属性处理源文件，并每次为map脚本提供一行文本。相反地，`TextOutputFormat`属性确保准确无误地把reduce任务的输出写入文本数据。当然，我们可以根据需要进行修改。

接下来，我们通过相当繁琐的命令行将Streaming作业提交到Hadoop。每个脚本文件之所以被指定了两次，是因为每个节点上无法使用的文件都必须由Hadoop打包并在集群内部流转，注意需要通过`-file`选项来指定。我们还需要告诉Hadoop，分别由哪个脚本文件担任mapper和reducer角色。

最后，我们查看作业的输出，它应当与前述基于Java实现WordCount的输出一致。

在作业中使用Streaming的区别

使用Streaming的WordCount mapper看起来比Java版本简单很多，但reducer的逻辑似乎更为复杂。这是为什么呢？原因是，当使用Streaming时，Hadoop和任务之间的隐含协议发生了变化。

在Java中，我们知道，对于每个输入键值对，`map()`方法都会被调用一次；对于每个键及其相应的一系列值，`reduce()`方法也会分别被调用。

在Streaming中，不再存在`map`或`reduce`方法的概念，我们只能自己编写脚本处理收到的数据流。这改变了用户编写reducer的方式。在Java中，Hadoop负责为每个键的对应值进行分组，每次调用`reduce`方法都会接收一个键和与之对应的所有值列表。在Streaming中，每个`reduce`任务的实例每次接收一个单独的未分组的值。

Hadoop Streaming对键进行了排序。例如，假如mapper输出如下数据：

```
First    1
Word     1
Word     1
A        1
First    1
```

Streaming reducer将以下列顺序接收这些数据：

```
A        1
First    1
First    1
Word     1
Word     1
```


Hadoop仍然收集每个键的值，并确保每个键只传递给一个reducer。换句话说，reducer得到若干键的所有值，它们已被组合在一起。然而，它们不会被打包到单个reducer执行，也就是说，每次执行一个键，与Java API中完全一样。

这就解释了Ruby reducer使用的机制：它首先为当前的词语设置空的默认值，然后在读取每行内容后，确定该行的键与前一行的键是否相同。如果答案是肯定的，累加该键的值。否则，保持前一个键的值不变，其最终结果被发送到标准输出，并开始为新词计数。

前几章我们一直在说Hadoop为我们做了多少多少工作，好像有多复杂似的。但只要写几个Streaming reducer，你就会发现实际上没那么复杂。同时请记住，Hadoop仍然负责向不同map任务分配split，以及将给定键的对应值发送给同一个reducer。这些行为可以通过配置更改mapper和reducer的数量来改变，就像使用Java API一样。

4.3　分析大数据集

有了使用Java和Streaming编写MapReduce作业的能力之后，现在我们将探讨一个比之前见过的任何数据集都更有意义的数据集。我们将讨论如何分析大数据集，以及Hadoop可以解答的关于大数据集的各种问题。

4.3.1　获取UFO目击事件数据集

我们将使用一个公共领域数据集，它包含了超过60 000次UFO目击事件的数据。这个数据集由InfoChimps托管在http://www.infochimps.com/datasets/60000-documented-ufo-sightings-with-text-descriptions-and-metada。

要下载这个数据集，需要注册一个免费的InfoChimps账号。

这些数据由一系列包含下列字段的UFO目击事件记录组成。

(1) **Sighting date**：UFO目击事件发生的时间。
(2) **Recorded date**：报告目击事件的日期，通常与目击事件时间不同。
(3) **Location**：目击事件发生的地点。
(4) **Shape**：UFO形状的简要描述，例如，菱形、发光体、圆筒状。
(5) **Duration**：目击事件的持续时间。
(6) **Description**：目击事件的大致描述。

下载完成后，读者会发现该数据有多种格式。我们将使用`.tsv`（制表符分隔值）版本。

4.3.2　了解数据集

当面临一个新数据集的时候，通常很难感知其性质、广度和涉及数据的质量。在着手后续分析之前，通常需要先搞清楚几个问题。

- 数据集有多大？
- 记录的完整性如何？
- 记录与我们期待的格式匹配程度如何？

第一个问题是一个简单的数据规模的问题。我们谈论的是数百、数千、数百万，或是更多的记录吗？第二个问题是询问数据的完整性。假如你希望每条记录有10个字段（如果是结构化或半结构化数据），那么有多少条记录的关键字段非空？最后一个问题在这一点上进行扩展，记录的表现形式与你所期望的数据格式和展示形式是否吻合？

4.4　实践环节：统计汇总 UFO 数据

现在我们有了这些数据，让我们先统计一下数据规模，以及多少条记录是不完整的。

(1) 以`ufo.tsv`为名在HDFS上保存UFO**制表符分隔值**（TSV）数据文件的同时，将下列内容保存为`summarymapper.rb`文件。

```
#!/usr/bin/env ruby

while line = gets
    puts "total\t1"
    parts = line.split("\t")
    puts "badline\t1" if parts.size != 6
    puts "sighted\t1" if !parts[0].empty?
    puts "recorded\t1" if !parts[1].empty?
    puts "location\t1" if !parts[2].empty?
```

```
        puts "shape\t1" if !parts[3].empty?
        puts "duration\t1" if !parts[4].empty?
        puts "description\t1" if !parts[5].empty?
    end
```

(2) 执行下列命令，使summarymapper.rb成为可执行文件。

```
$ chmod +x summarymapper.rb
```

(3) 使用Streaming执行作业，如下所示。

```
$ hadoop jar hadoop/contrib/streaming/hadoop-streaming-1.0.3.jar
-file summarymapper.rb -mapper summarymapper.rb -file wcreducer.rb
-reducer wcreducer.rb -input ufo.tsv -output ufosummary
```

(4) 获取汇总数据。

```
$ hadoop fs -cat ufosummary/part-0000
```

4

原理分析

请记住，我们的UFO目击事件应该有前述的6个字段。它们分别是：

- 目击事件的日期
- 报告目击事件的日期
- 目击事件的地点
- UFO的形状
- 目击事件的持续时间
- 事件的大致描述

mapper处理文件，除识别潜在的不完整记录，还统计了记录总数。

在处理文件时，我们通过简单地计数遇到多少不同的记录，得到了记录总数。有一些记录要么少于6个字段，要么至少有一个字段的值为空，在处理文件时通过标记这些记录来识别出潜在的不完整记录。

因此，mapper在完成文件处理任务时，读取每一行并完成了以下3项工作。

- 输出已被处理的记录总数，该值随着程序运行不断递增。
- 以制表符来分隔记录，并输出少于6个字段的记录总数。
- 对6个预期字段，mapper输出字段值非空（也就是该字段有数据）的字段数，尽管这与数据质量没有关系。

我们故意把mapper编写成输出(token, count)的形式。这样就能够使用以前实现的WordCount reducer作为这个作业的reducer。当然，还有更高效的实现方式，但考虑到这个作业不可能被频繁地执行，为了方便，这样做是值得的。

在写作本书时，本作业的输出结果如下所示。

```
badline324
description61372
duration58961
location61377
recorded61377
shape58855
sighted61377
total61377
```

从这些数字我们可以看出，共有61 300条记录。这些记录都提供了目击事件发生的时间、报告时间，以及目击事件发生地这些字段的值。大约58 000~59 000条记录有形状和持续时间的值，几乎所有记录都有相关描述。

当用制表符分割时，我们发现有372行记录的字段不足6个。然而，因为只有5条记录的“描述”字段没有值，这表明不完整记录中的制表符不是太少，而是太多了。当然，我们可以基于这个事实调整mapper以收集详细信息。这可能是由文本描述使用的制表符所致。不过就眼下的分析而言，我们姑且认为大多数记录都有6个字段，而且每个字段都有值好了。至于每条记录中是不是有多余的制表符，先不管了。

调查UFO形状

在这些报告的所有字段中，形状是最直接引起我们兴趣的，因为基于该字段信息，我们可以对数据采取一些有意思的分组方式。

4.5　实践环节：统计形状数据

刚才统计了UFO数据集的记录总数，现在让我们对UFO形状数据进行针对性的统计。

(1) 将下列代码保存为shapemapper.rb文件。

```
#!/usr/bin/env ruby

while line = gets
    parts = line.split("\t")
    if parts.size == 6
        shape = parts[3].strip
        puts shape+"\t1" if !shape.empty?
    end
end
```

(2) 为shapemapper.rb文件增加可执行权限。

```
$ chmod +x shapemapper.rb
```

(3) 再次使用WordCount reducer执行这个作业。

```
$ hadoop jar hadoop/contrib/streaming/hadoop-streaming-1.0.3.jarr
--file shapemapper.rb -mapper shapemapper.rb -file wcreducer.rb
-reducer wcreducer.rb -input ufo.tsv -output shapes
```

(4) 获取形状数据信息。

```
$ hadoop fs -cat shapes/part-00000
```


原理分析

本例使用的mapper相当简单。它把每条记录按其组成字段分开，丢弃不足6个字段的记录，使用计数器统计形状字段非空的记录数并输出该值。

出于本节目的，我们乐于忽略任何与期望的规则不匹配的记录。也许一次UFO目击事件的记录将彻底证明UFO是确实存在的，但即便如此，一次记录不可能对我们的分析有太大影响。在决定丢弃一些记录之前，应仔细考虑一下不同记录的潜在可能值。如果读者投身于更关注某种变化趋势的聚类工作，那么个别记录可能无足轻重。但是在个别值可能严重影响分析结论或者必须考虑在内的情况下，不能弃用这些记录，最好试着小心翼翼地解析并恢复它们。我们将在第6章中对这种权衡进行更多的讨论。

像往常一样，将mapper设置可执行属性并运行生成的作业后，结果表明有29种不同的UFO形状。为节省版面，这里以制表符分割的紧凑结构展示作业的输出。

```
changed1 changing1533 chevron758 cigar1774
circle5250 cone265 crescent2 cross177
cylinder981 delta8 diamond909 disk4798
dome1 egg661 fireball3437 flare1
flash988 formation1775 hexagon1 light12140
other4574 oval2859 pyramid1 rectangle957
round2 sphere3614 teardrop592 triangle6036
unknown4459
```

正如我们所看到的，不同形状的目击事件发生的频率相差很大。某些形状的UFO目击事件只出现过1次，如pyramid（金字塔），而light（发光体）出现的频率超过了全部目击事件的1/5。考虑到许多UFO目击事件发生在夜晚，可能有人认为发光体的描述并不是非常有用，或者不够具体。如果再算上other（其他）和unknown（未知），我们看到58 000次形状报告中约有21 000个实际上可能没有任何用处。我们并不打算刨根究底或者做其他研究，所以到底有多少次目击事件的形状是有用的这一点并不十分重要。但重要的是，我们要开始从这些角度考虑数据。甚至通过这些类

型的统计分析，我们能够洞察数据本质和分析结论的质量。例如，我们已经发现在61 000次目击事件中，只有58 000次事件报告的形状字段非空，而这其中又有21 000次目击事件的形状字段是含糊值。这样，我们就知道61 000次目击事件的样本集只提供了37 000个可能开展工作的形状报告。如果你的分析结论建立在少量样本基础之上，那么一定要进行这种前期统计以确定数据集是否能满足实际需要。

4.6 实践环节：找出目击事件的持续时间与 UFO 形状的关系

让我们稍加详细地分析形状数据。我们想知道，在目击事件的持续时间与UFO形状之间是否存在某种关联。也许雪茄状UFO比其他形状UFO持续的时间更长，又或许UFO编队的持续时间总是固定的。

(1) 保存下列内容为shapetimemapper.rb文件。

```
#!/usr/bin/env ruby

pattern = Regexp.new / \d* ?((min)|(sec))/

while line = gets
parts = line.split("\t")
if parts.size == 6


shape = parts[3].strip
duration = parts[4].strip.downcase
if !shape.empty? && !duration.empty?
match = pattern.match(duration)
time = / \d*/.match(match[0])[0]
unit = match[1]
time = Integer(time)
time = time * 60 if unit == "min"
if unit == "min"
puts shape+"\t"+time.to_s
end
end
end
```

(2) 通过执行以下命令把shapetimemapper.rb文件做成可执行文件。

```
$ chmod +x shapetimemapper.rb
```

(3) 保存下列内容为shapetimereducer.rb文件。

```
#!/usr/bin/env ruby

current = nil
min = 0
max = 0
mean = 0
```

```
total = 0
count = 0

while line = gets
word, time = line.split("\t")
time = Integer(time)

if word == current
count = count+1
total = total+time
min = time if time < min
max = time if time > max
else
puts current+"\t"+min.to_s+" "+max.to_s+" "+(total/count).to_s if
current
current = word
count = 1
total = time
min = time
max = time
end
end
puts current+"\t"+min.to_s+" "+max.to_s+" "+(total/count).to_s
```

(4) 通过执行以下命令把shapetimereducer.rb做成可执行文件。

```
$ chmod +x shapetimereducer.rb
```

(5) 运行作业。

```
$ hadoop jar hadoop/contrib/streaminghHadoop-streaming-1.0.3.jar
-file shapetimemapper.rb -mapper shapetimemapper.rb -file
shapetimereducer.rb -reducer shapetimereducer.rb -input ufo.tsv
-output shapetime
```

(6) 获取结果。

```
$ hadoop fs -cat shapetime/part-00000
```

原理分析

由于持续时间字段的原因，mapper比之前的例子稍微复杂一些。快速浏览一些示例记录，我们发现部分记录的持续时间字段值如下。

```
15 seconds
2 minutes
2 min
2minutes
5-10 seconds
```

换句话说，持续时间字段的值既可能是时间区间，也可能是绝对的时间值，而且其格式不同，

时间单位不一致。为简单起见，我们决定对数据进行有限解释：如果存在绝对的时间值，我们就采用该值，否则采用时间区间的上限作为某次UFO目击事件的持续时间。假设时间单位用`min`或者`sec`字符串表示，并将所有的时间都转为以秒为单位。借助一些正则表达式，我们依照上述规则分解持续时间字段的值，并将其转换为以秒为单位。请再次注意，我们简单地弃用那些与预期规则不一致的记录，当然这个做法不一定总是合适的。

遵循与之前示例中的reducer相同的模式，从一个默认键开始读取该键的值，直到遇到一个新的键。在这种情况下，我们想要获取每种形状的UFO持续时间的极小值、极大值、平均值，因此需要使用大量变量来跟踪所需数据。

请记住，Streaming reducer需要处理多个键及与每个键对应的一系列值，当新一行的键发生变化时，reducer需要识别这个变化，因此，需要标识已处理的前一个键的最后一个值。与此相反，Java reducer更简单一些，每次执行中仅处理一个键的关联值。

把mapper和reducer都做成可执行文件之后，运行本作业并获得如下结果。这些结果中除去了目击次数少于10次的UFO形状，而且为节省空间，输出更为紧凑。紧跟每个形状的几个数字分别是其持续时间的最小值、最大值以及平均值。

```
changing0 5400 670 chevron0 3600 333
cigar0 5400 370 circle0 7200 423
cone0 4500 498 cross2 3600 460
cylinder0 5760 380 diamond0 7800 519
disk0 5400 449 egg0 5400 383
fireball0 5400 236 flash0 7200 303
formation0 5400 434 light0 9000 462
other0 5400 418 oval0 5400 405
rectangle0 4200 352 sphere0 14400 396
teardrop0 2700 335 triangle0 18000 375
unknown0 6000 470
```

令人惊奇的是，所有形状的平均持续时间的变化范围相对较小。大多数UFO目击事件的平均持续时间在350~430秒之间。而且有趣的是，我们也看到，fireball（火球）的平均持续时间最短，changing（变化不定）的平均持续时间最长，这两者在某种程度上都符合人的直觉。火球嘛，当然不会持续太久，而变化不定的物体可能需要较长时间人们才能注意到它的变化。

在Hadoop外部使用Streaming脚本

最后一个例子的mapper和reducer更为复杂，它很好地说明了使用Streaming帮助MapReduce开发的另一种方法：在Hadoop外部执行Streaming脚本。

通常，在MapReduce开发过程中使用产品数据示例来测试代码是一种较好的做法。但在HDFS上使用Java编写map和reduce任务使调试问题或改进复杂逻辑变得很困难。而使用从命令行读取输入的map和reduce任务，读者可以直接用一些数据测试它们，快速从结果中得到反馈。如果读者

的开发环境可以集成Hadoop或者在独立模式下使用Hadoop，问题是最少的。记住，Streaming允许用户在Hadoop外部使用脚本，没准哪一天你就能用上这个技术。

在开发这些脚本的时候，作者注意到，在他使用的UFO数据文件中，最后一组记录的数据结构化程度要优于文件开头部分数据。使用mapper执行快速测试只需使用如下命令：

```
$ tail ufo.tsv | shapetimemapper.rb
```

这个方法可被应用到使用map和reduce脚本的整个工作流程。

4.7 实践环节：在命令行中执行形状/时间分析

通过本地命令行的方式执行这种数据分析的方法可能不是很明显，因此我们先看一个例子。

对存储在本地文件系统的UFO数据文件，执行下列命令：

```
$ cat ufo.tsv | shapetimemapper.rb | sort| shapetimereducer.rb
```

原理分析

短短的一句Unix命令行产生的输出，却与之前完整的MapReduce作业的输出一模一样，简直不可思议。详细分析上述命令所执行的操作，就会明白其原因。

首先，将输入文件按行发送给mapper，每次一行。这个步骤的输出被传到Unix sort工具，经过排序的输出又被按行传给reducer。当然，这是常用MapReduce作业流程的简化表示。

那么，显而易见的问题是，如果我们能使用命令行进行等效分析，为什么还要用Hadoop呢？答案依然照旧——大数据处理。这种简单的方法可以有效处理类似UFO目击事件这样大小的数据文件，这个文件虽不算小，也只有71 MB。今天，随便找一块硬盘就可以容纳数千份这样的数据集。

那么，如果数据集大小是71 GB，甚至是71 TB，该怎么办呢？在后一种情况下，至少需要将数据分布在多台主机，然后决定如何切分数据，组合部分答案，并且处理遇到的不可避免的故障。换句话说，这时候就需要类似Hadoop这样的工具了。

然而，也不要小看使用类似的命令行工具，这些方法应当在MapReduce开发过程中得到良好使用。

使用Java分析形状和地点

现在，我们考虑用Java MapReduce API对UFO目击事件报告中的形状和地点数据进行分析。

在开始编写代码之前，回想一下前面是如何分析UFO目击事件数据集的各个字段的。之前的

mapper有一个通用的模式。

- 丢弃已损坏记录。
- 处理有效记录，提取感兴趣的字段。
- 针对这些记录，输出我们所关注的数据。

现在，假如我们打算编写Java mapper来分析UFO目击事件发生的地点，之后可能分析报告时间，我们将遵循类似的模式。那么，我们是否可以避免由此产生的代码重复问题呢?

答案是肯定的，通过使用`org.apache.hadoop.mapred.lib.ChainMapper`即可解决该问题。`ChainMapper`类可以顺序执行多个mapper，而且最后的mapper输出会传递给reducer。`ChainMapper`不仅适用于这种数据清理，而且在分析特定作业时，先通过它执行多个map型任务再应用reducer的做法也很常见。

这种做法需要编写一个校验mapper，它可用于将来所有的字段分析作业。校验mapper会丢弃错误记录行，只将有效的行传入实际的业务逻辑mapper。这样的话，目前的业务逻辑mapper就可以专注于分析数据而不用担心粗粒度的校验。

另一种可供选择的做法是，在自定义的`InputFormat`类中验证数据并丢弃无效记录。哪一种方法更合理取决于用户的特定情况。

链里的每个mapper都会在一个独立的JVM中执行，所以不必担心使用多个mapper会增加文件系统的I/O负载。

4.8 实践环节：使用 ChainMapper 进行字段验证/分析

让我们在作业中使用`ChainMapper`类验证记录是否有效。

(1) 在`UFORecordValidationMapper.java`文件中创建如下类。

```
import java.io.IOException;

import org.apache.hadoop.io.* ;
import org.apache.hadoop.mapred.* ;
import org.apache.hadoop.mapred.lib.* ;

public class UFORecordValidationMapper extends MapReduceBase
implements Mapper<LongWritable, Text, LongWritable, Text>
{

   public void map(LongWritable key, Text value,
       OutputCollector<LongWritable, Text> output,
       Reporter reporter) throws IOException
{
String line = value.toString();
```

```
    if (validate(line))
        output.collect(key, value);
}

    private boolean validate(String str)
    {
        String[] parts = str.split("\t") ;

        if (parts.length != 6)
        return false ;

        return true ;
    }
}
```

(2) 使用下列代码创建`UFOLocation.java`文件。

```
import java.io.IOException;
import java.util.Iterator ;
import java.util.regex.* ;

import org.apache.hadoop.conf.* ;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.* ;
import org.apache.hadoop.mapred.* ;
import org.apache.hadoop.mapred.lib.* ;

public class UFOLocation
{

    public static class MapClass extends MapReduceBase
implements Mapper<LongWritable, Text, Text, LongWritable>
{

private final static LongWritable one = new LongWritable(1);
private static Pattern locationPattern = Pattern.compile(
"[a-zA-Z]{2}[^a-zA-Z]*$") ;

public void map(LongWritable key, Text value,
OutputCollector<Text, LongWritable> output,
Reporter reporter) throws IOException
{
String line = value.toString();
        String[] fields = line.split("\t") ;
        String location = fields[2].trim() ;


      if (location.length() >= 2)
      {

          Matcher matcher = locationPattern.matcher(location) ;
          if (matcher.find() )
```

```
            {
                int start = matcher.start() ;
                String state = location.substring(start,start+2);

                output.collect(new Text(state.toUpperCase()), One);
            }
        }
    }
}

public static void main(String[] args) throws Exception
{
    Configuration config = new Configuration() ;
JobConf conf = new JobConf(config, UFOLocation.class);
conf.setJobName("UFOLocation");

conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(LongWritable.class);

JobConf mapconf1 = new JobConf(false) ;
ChainMapper.addMapper( conf, UFORecordValidationMapper.class,
LongWritable.class, Text.class, LongWritable.class,
Text.class, true, mapconf1) ;

JobConf mapconf2 = new JobConf(false) ;
ChainMapper.addMapper( conf, MapClass.class,
LongWritable.class, Text.class,
Text.class, LongWritable.class, true, mapconf2) ;
conf.setMapperClass(ChainMapper.class);
conf.setCombinerClass(LongSumReducer.class);
conf.setReducerClass(LongSumReducer.class);

FileInputFormat.setInputPaths(conf,args[0]) ;
FileOutputFormat.setOutputPath(conf, new Path(args[1])) ;

JobClient.runJob(conf);
}
}
```

(3) 编译上述两个文件。

```
$ javac UFORecordValidationMapper.java UFOLocation.java
```

(4) 将上述两个类文件打包为`ufo.jar`文件并提交作业至Hadoop。

```
$ Hadoop jar ufo.jar UFOLocation ufo.tsv output
```

(5) 将输出文件拷贝至本地文件系统并检查它。

```
$ Hadoop fs -get output/part-00000 locations.txt
$ more locations.txt
```

原理分析

上述代码实现了很多功能，让我们一段一段地进行分析。

第一个mapper是一个简单的验证mapper。该类与标准MapReduce API的接口相同，map返回了验证的结果。我们将验证方法分割为单独的方法以突出mapper的功能，但是这些校验可以很容易地在map主方法中实现。为了简单起见，我们坚持以前的验证策略，检查字段数并丢弃那些少于6个字段的记录。

请注意，ChainMapper类不幸地成为最后迁移到context对象API的组件之一，截至Hadoop 1.0，只能使用旧的API。ChainMapper仍是一个有效的概念和有用的工具，但直到Hadoop 2.0，它才会将被迁移到org.apache.hadoop.mapreduce.lib.chain包，所以目前仍需使用其较老的方法。

另一个文件实现了另一个mapper，并在主方法中包含了一个升级后的驱动。该mapper在UFO目击事件报告的地点字段寻找双字母序列。手工检查数据发现，很明显大部分地点字段的值是“城市，州”的形式，在这里，标准的双字符缩写表示UFO目击事件发生的州名。

然而，有些记录加入了后括号、句号或其他标点符号。另一些记录根本就不是这种格式。对我们而言，我们很乐意抛弃那些记录，并专心处理那些我们比较看重的、以双字符的州名缩写结尾的记录。

map方法使用另一个正则表达式从地点字段中提取州名缩写，并输出其大写形式和计数结果。

本作业的驱动程序发生的变化最大。之前的驱动配置中仅包含一个map类，而本作业的驱动配置要多次调用ChainMapper类。

在驱动中多次调用ChainMapper类的一般模式是，为每个mapper新建一个配置对象，然后将mapper添加到ChainMapper类，同时指定输入和输出位置，并引用整个作业的配置对象。

请注意，上述两个mapper的参数略有不同。它们都输入LongWritable类型的键及Text类型的值。区别在于输出的数据类型：UFORecordValidationMapper输出LongWritable类型的键及Text类型的值，而UFOLocationMapper则相反，输出Text类型的键及LongWritable类型的值。

重要的是，要保证mapper链条末端的UFOLocationMapper的输入与reduce类（LongSumReducer）的输入类型相匹配。在使用ChainMapper类时，只要符合下列条件，链条中的mapper可以有不同的输入和输出：

- 除最后一个mapper外，链条中每个map的输出与下一个mapper的输入相匹配；
- 对最后一个mapper而言，其输出与reducer输入相匹配。

我们编译这些类并将其打包到同一个jar文件。这是我们第一次从多个Java源文件获得捆绑输出。正如所料，这没什么神奇的。jar文件、路径以及类名的常用规则在这里同样适用。因为在这个例子中，所有类都在同一个jar包中，不必担心驱动类文件的引入。

随后，我们运行MapReduce作业并检查输出，结果与预期有些不同。

一展身手

使用Java API和前面的ChainMapper例子来重新实现之前用Ruby编写的mapper，该mapper输出不同UFO形状出现的频率与其持续时间。

1. 缩写太多

前一节作业的输出结果如下所示。

```
AB    286
AD    6
AE    7
AI    6
AK    234
AL    548
AM    22
AN    161
...
```

该文件有186个不同的双字符条目。简单地说，从地点字段提取双字符的方法并非完全可行。

在人工分析源文件之后，许多数据问题浮出水面。

- 州名的大写缩写不一致。
- 大量的目击事件并非发生在美国，尽管它们可能遵守类似（城市，地区）的格式，但是它们的缩写并不在我们预计的50个地区缩写之内。
- 有些字段根本不遵守（城市，地区）这样的规则，但仍会被正则表达式采集到。

我们需要对结果进行过滤，最好是将美国记录标准化为正确的州名输出，并将其余数据划分为一个范围更宽泛的大类。

为了执行这个任务，需要在mapper中添加一些内容，让它明白什么是有效的美国州名缩写。当然，我们可以将其硬编码到mapper中，但这似乎不是正确的做法。虽然现在我们计划将所有非美国的目击事件视为一类，但我们今后还可能对该类进行扩展，比如按国家进行划分。如果将州名缩写硬编码到mapper，那就需要每次重新编译我们的mapper。

2. 使用分布式缓存

为了使作业的所有任务共享引用数据，Hadoop为我们提供了一种可供选择的机制——分布

式缓存。它可以把map或reduce任务要用的通用只读文件在所有节点之间共享。这些文件可以是像本例中的文本数据，也可以是其他jar文件、二进制数据或一些归档文件等，任何文件都可以。

要被共享的文件放置在HDFS，作业驱动将其加入到DistributedCache。在作业执行前，Hadoop的每个节点会将文件复制到本地文件系统，这意味着每个任务对文件都具有本地访问权限。

另一种方法是将需要的文件捆绑编译成作业jar，然后提交给Hadoop。将数据打包到jar文件使数据很难在作业间共享，而且如果数据发生变化，还需要重新编译jar文件。

4.9 实践环节：使用 Distributed Cache 改进地点输出

我们现在使用Distributed Cache在集群内部共享美国州名及其缩写列表。

(1) 在本地文件系统创建名为states.txt的数据文件。文件中的每行内容都是一个以制表符分隔的州名缩写及全称。读者可以从本书主页获取该文件。该文件的开头部分如下所示。

```
AL    Alabama
AK    Alaska
AZ    Arizona
AR    Arkansas
CA    California

...
```

(2) 将states.txt文件放到HDFS上。

```
$ hadoop fs -put states.txt states.txt
```

(3) 将之前的UFOLocation.java拷贝为UFOLocation2.java文件，并通过添加下列“引用”声明修改该文件。

```
import java.io.* ;
import java.net.* ;
import java.util.* ;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.filecache.DistributedCache ;
```

(4) 将下列内容添加到驱动程序的主方法中，放在设置作业名的代码之后。

```
DistributedCache.addCacheFile(new URI ("/user/hadoop/states.txt"), conf) ;
```

(5) 将map类替换为以下内容。

```
    public static class MapClass extends MapReduceBase
implements Mapper<LongWritable, Text, Text, LongWritable>
    {

        private final static LongWritable one = new
```

```
    LongWritable(1);
            private static Pattern locationPattern =
  Pattern.compile( "[a-zA-Z]{2}[^a-zA-Z]*$") ;
            private Map<String, String> stateNames ;

            @Override


            public void configure( JobConf job)
            {

                try
                {
                    Path[] cacheFiles = DistributedCache.
    getLocalCacheFiles(job) ;
                    setupStateMap( cacheFiles[0].toString()) ;
                } catch (IOException e)
    {
    System.err.println("Error reading state file.") ;
                        System.exit(1) ;
    }
            }

            private void setupStateMap(String filename)
    throws IOException
            {
               Map<String, String> states = new  HashMap<String,
    String>();
               BufferedReader reader = new BufferedReader( new
    FileReader(filename)) ;
                String line = reader.readLine() ;
                while (line != null)
                {
                   String[] split = line.split("\t") ;
                   states.put(split[0], split[1]) ;
                   line = reader.readLine() ;
                }

                stateNames = states ;
            }

            public void map(LongWritable key, Text value,
                OutputCollector<Text, LongWritable> output,
                Reporter reporter) throws IOException
            {
                String line = value.toString();
            String[] fields = line.split("\t") ;
            String location = fields[2].trim() ;
            if (location.length() >= 2)
            {

                Matcher matcher = locationPattern.matcher(location) ;
                if (matcher.find() )
```

```
            {
                int start = matcher.start() ;
                String state = location.substring(start, start+2)
;
                
                output.collect(newText(lookupState(state.
toUpperCase())), one);
            }
        }
    }
    
    private String lookupState( String state)
    {
        String fullName = stateNames.get(state) ;
        
        return fullName == null? "Other": fullName ;
        }
}
```


(6) 编译这些类并将作业提交至Hadoop。随后获取结果文件。

原理分析

首先，我们创建了一个本作业要用到的查询文件，并将其放在HDFS上。要添加到Distributed Cache的文件必须首先拷贝到HDFS文件系统。

在创建新的作业文件之后，我们添加了必需的类引用。然后，修改驱动类，将我们想在每个节点上访问的文件添加到Distributed Cache。文件名可以用多种方式指定，但最简单的方法是使用该文件在HDFS上的绝对路径。

我们也对mapper类进行了大量修改。其中，新增了一个重写的`configure`方法，该方法使用map方法将州名缩写与全称关联起来。

`configure`方法在任务启动时被调用，其默认实现不执行任何操作。在我们重写的版本中，该方法获取已添加到Distributed Cache中的文件阵列。因为我们知道缓存中只有一个文件，所以就直接取得数组中第一项，并将其传给`utility`方法，该方法解析文件并使用文件内容填充州名缩写查询map。请注意，一旦获得了文件引用，我们就可以使用标准Java I/O类来访问该文件了，毕竟它只是一个本地文件系统的文件。

我们添加了另一种方法来执行查询，该方法的参数是从地点字段获取的字符串，如果有匹配结果，它返回州名的全称，否则返回字符串`Other`。该方法在`OutputCollector`类对map方法的结果进行写操作之前调用。

本作业的结果大致如下。

```
Alabama       548
Alaska        234
Arizona       2097
Arkansas      534
California    7679
...
Other         4531...

...
```

这段代码运行得很好，但我们在此期间却丢失了一些信息。在验证mapper中，我们丢掉了少于6个字段的所有记录。虽然我们不在意丢弃个别记录，但我们可能会关心被丢弃的记录数量是不是很多。目前，确定被丢弃记录数量的唯一方法是，累计包含有效地点的记录数并从文件记录总数中减去该值。我们也可以尝试将这些数据传入作业的其余部分，以一个特殊的reduce键来收集该值，但这个想法似乎行不通。幸运的是，我们有一个更好的办法。

4.10 计数器、状态和其他输出

每个MapReduce作业结尾都有一些与计数器相关的输出，比如下面这些：

```
12/02/12 06:28:51 INFO mapred.JobClient: Counters: 22
12/02/12 06:28:51 INFO mapred.JobClient:   Job Counters
12/02/12 06:28:51 INFO mapred.JobClient:     Launched reduce tasks=1
12/02/12 06:28:51 INFO mapred.JobClient:     Launched map tasks=18
12/02/12 06:28:51 INFO mapred.JobClient:     Data-local map tasks=18
12/02/12 06:28:51 INFO mapred.JobClient:   SkippingTaskCounters
12/02/12 06:28:51 INFO mapred.JobClient:     MapProcessedRecords=61393
...
```

你也可以添加自定义计数器，从所有任务中聚合信息，并在最终输出和MapReduce web UI中得到统计结果。

4.11 实践环节：创建计数器、任务状态和写入日志

我们将修改`UFORecordValidationMapper`，用它统计被略去的记录数，并标记一些记录作业信息的其他措施。

(1) 创建`UFOCountingRecordValidationMapper.java`文件，输入以下内容。

```
import java.io.IOException;

import org.apache.hadoop.io.* ;
import org.apache.hadoop.mapred.* ;
import org.apache.hadoop.mapred.lib.* ;
```

```
public class UFOCountingRecordValidationMapper extends
MapReduceBase
implements Mapper<LongWritable, Text, LongWritable, Text>
{

    public enum LineCounters
    {
        BAD_LINES,
        TOO_MANY_TABS,
        TOO_FEW_TABS
    } ;

    public void map(LongWritable key, Text value,
        OutputCollector<LongWritable, Text> output,
        Reporter reporter) throws IOException
    {
        String line = value.toString();

        if (validate(line, reporter))
Output.collect(key, value);
     }

    private boolean validate(String str, Reporter reporter)
    {
        String[] parts = str.split("\t") ;

        if (parts.length != 6)
        {
            if (parts.length < 6)
            {

reporter.incrCounter(LineCounters.TOO_FEW_TABS, 1) ;
            }
            else
            {
                reporter.incrCounter(LineCounters.TOO_MANY_TABS,
1) ;
             }
             reporter.incrCounter(LineCounters.BAD_LINES, 1) ;

if((reporter.getCounter(
LineCounters.BAD_LINES).getCounter()%10)
== 0)
            {
                reporter.setStatus("Got 10 bad lines.") ;
                System.err.println("Read another 10 bad lines.") ;
            }

            return false ;
        }
        return true ;
    }
        }
```

(2) 复制`UFOLocation2.java`，另存为`UFOLocation3.java`，使用下列新mapper代替`UFORecordValidationMapper`。

```
...
        JobConf mapconf1 = new JobConf(false) ;
        ChainMapper.addMapper( conf,
UFOCountingRecordValidationMapper.class,
            LongWritable.class, Text.class, LongWritable.class,
Text.class,
            true, mapconf1) ;
```

(3) 编译文件，打包为jar并提交作业至Hadoop。

```
...
12/02/12 06:28:51 INFO mapred.JobClient: Counters: 22
12/02/12 06:28:51 INFO mapred.JobClient: UFOCountingRecordValida
tionMapper$LineCounters
12/02/12 06:28:51 INFO mapred.JobClient:     TOO_MANY_TABS=324
12/02/12 06:28:51 INFO mapred.JobClient:     BAD_LINES=326
12/02/12 06:28:51 INFO mapred.JobClient:     TOO_FEW_TABS=2
12/02/12 06:28:51 INFO mapred.JobClient:     Job Counters
```

(4) 使用浏览器打开MapReduce web UI（默认情况下，MapReduce web UI是在JobTracker主机的50030端口）。选择**Completed Jobs**列表底部的作业，会看到类似下面的屏幕截图。

Hadoop job_201202111619_0023 on head - Windows Internet Explorer

Hadoop job_201202111619_0023 on head

User: hadoop
Job Name: UFOLocation
Job File: hdfs://head:9000/var/hadoop/mapred/system/job_201202111619_0023/job.xml
Job Setup: Successful
Status: Succeeded
Started at: Sun Feb 12 15:00:35 PST 2012
Finished at: Sun Feb 12 15:01:55 PST 2012
Finished in: 1mins, 19sec
Job Cleanup: Successful

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	18	0	0	18	0	0 / 2
reduce	100.00%	1	0	0	1	0	0 / 0

	Counter	Map	Reduce	Total
UFOCountingRecordValidationMapper$LineCounters	TOO_MANY_TABS	324	0	324
	TOO_FEW_TABS	2	0	2
	BAD_LINES	326	0	326
	Launched reduce tasks	0	0	1

(5) 点击map任务的链接，你应当看到一个概述页面，如下图所示。

Hadoop map task list for job_201202111619_0023 on head - Windows Internet Explorer

Hadoop map task list for job_201202111619_0023 on head

Completed Tasks

Task	Complete	Status	Start Time	Finish Time	Errors
task_201202111619_0023_m_000000	100.00%	Got 10 bad lines.	12-Feb-2012 15:00:40	12-Feb-2012 15:00:49 (9sec)	
task_201202111619_0023_m_000001	100.00%	Got 10 bad lines.	12-Feb-2012 15:00:40	12-Feb-2012 15:00:49 (9sec)	
task_201202111619_0023_m_000002	100.00%	Got 10 bad lines.	12-Feb-2012 15:00:41	12-Feb-2012 15:01:26 (44sec)	
task_201202111619_0023_m_000003	100.00%	hdfs://head:9000/user/hadoop/ufo.tsv:12582912+4194304	12-Feb-2012 15:00:41	12-Feb-2012 15:01:26 (44sec)	

(6) 选择一个带有自定义状态消息的任务，点击其计数器链接，结果应当与下图类似。

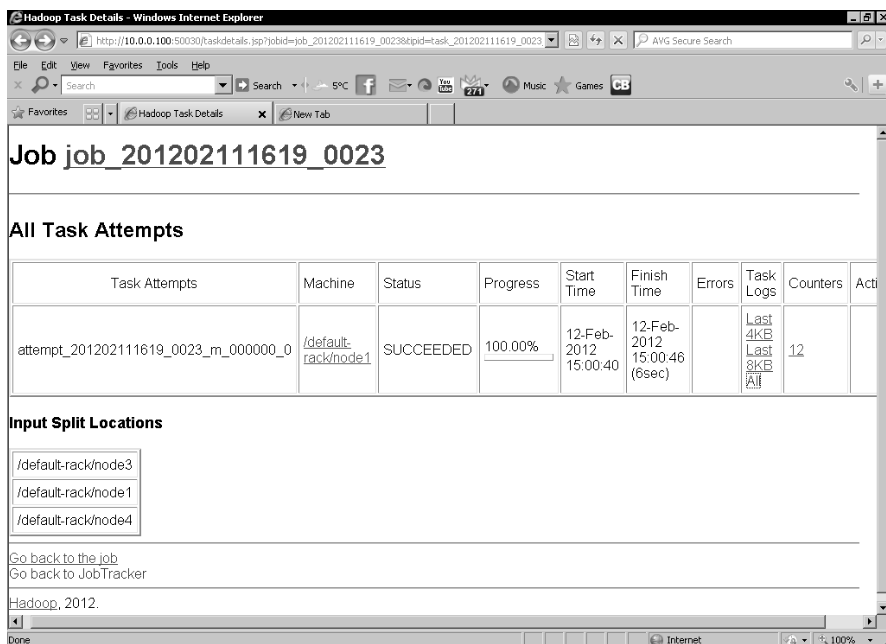
Counters for task_201202111619_0023_m_000000 - Windows Internet Explorer

Counters for task_201202111619_0023_m_000000

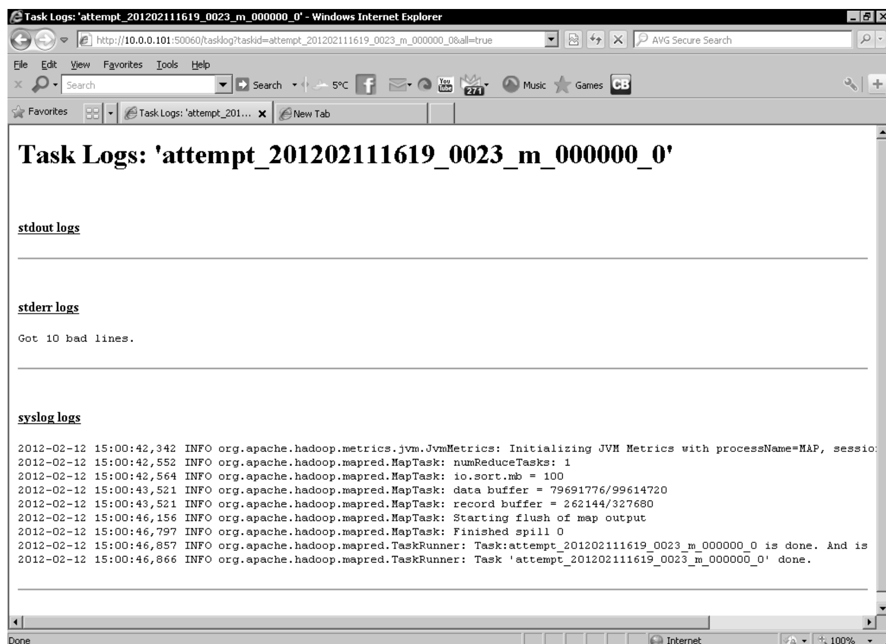
UFOCountingRecordValidationMapper$LineCounters	
TOO_MANY_TABS	19
BAD_LINES	19
SkippingTaskCounters	
MapProcessedRecords	4,411
FileSystemCounters	
HDFS_BYTES_READ	4,198,400
FILE_BYTES_WRITTEN	1,065
Map-Reduce Framework	
Combine output records	52
Map input records	4,411
Spilled Records	52
Map output bytes	76,359
Map input bytes	4,195,420
Combine input records	4,386
Map output records	4,386

Go back to the job
Go back to JobTracker

(7) 返回任务列表并点击任务ID，获得任务综述，结果与下图类似。



(8) 可在**Task Logs**列选择想要显示的数据量。点击**All**链接，结果如下图所示。



(9) 现在，登录到一个任务节点并浏览hadoop/logs/userlogs路径下的文件。每个任务都有一个专用目录，该目录下有若干文件。我们要查看的是stderr。

原理分析

为了添加新计数器，我们需要做的第一件事是创建一个承载它们的标准Java枚举。本例中，我们创建了一个**LineCounters**，Hadoop将其视为计数器组。在LineCounters中，有3个计数器分别对不符合数据格式的总行数，以及更细力度地对少于或多于6个字段的行的数量进行统计。这就是创建新计数器集合所需做的所有事情——定义枚举类型。计数开始后，Hadoop框架会自动维护计数器的值。

为了将中标数据加入到计数器，我们通过Reporter对象增加计数器的值。本例中，每次我们遇到错误数据行，少于6个字段的行或者多于6个字段的行时，分别将相应计数器的值加1。

我们也获取BAD_LINE计数器的值，假如它是10的倍数，执行下列任务：

- 设置任务状态以反映实际情况；
- 使用标准的Java System.err.println机制向stderr写入类似信息。

随后，我们转向MapReduce UI，验证是否可在作业概览中看到计数器总数，是否可在任务列表中看到带有自定义状态消息的所有任务。

之后，我们浏览网页用户接口，查看个别作业的计数器。对于我们在详细页面看到的任务，可以点击查看任务日志文件。

查看其中一个节点，我们发现，Hadoop为每个任务留存了日志，存放在文件系统的{HADOOP_HOME}/logs/userlogs目录下。在每个任务的子目录下，有标准流和普通任务日志的文件。如你所见，忙碌节点留下了大量的任务日志目录，从中找出我们感兴趣的任务目录不是一件容易的事。事实证明，使用网页接口查阅此类数据更为高效。



假如你用的是Hadoop context对象API，就要用Context.getCounter().increment()方法访问计数器。

信息太多

虽然不必再考虑如何获取作业状态和其他信息，但似乎突然间出现了太多令人困惑的信息可供选择。问题的实质是，在全分布式模式下，数据散布在各个节点，因此我们无法采用传统的单

击调试方法调试MapReduce作业。Ruby Streaming任务可轻易在命令行下运行，命令行的输出为调试问题提供了帮助；而使用Java语言实现的任务却无法模拟该行为。因此，开发者需要特别考虑，调试程序可能会用到哪些作业运行时信息。这应当包括一般性的作业运行状态，也应当包括可以为进一步调查问题提供帮助的其他信息。

计数器、任务状态消息以及老式Java日志可以协同工作。假如你对某种状况比较关注，将其设为计数器，每次发生这种状况时计数器都会记录。同时在发生这种状况时，设置任务状态消息。如果任务运行过程中产生了一些特殊数据，将其写入到`stderr`。由于很容易看到计数器的值，你可以在作业完成之后迅速得知你所关注的情况是否发生过。而且，在网页用户界面中，一眼就能看出你所关注的情况发生在哪个任务运行过程中。并且，你可以在网页用户界面中点击查看该任务的详细日志。

其实，你不需要等到整个作业完成后才开始调查。随着作业的执行，计数器和任务状态信息会在网页用户界面实时更新，所以只要计数器或任务状态信息提醒你出现了关注的情况，你就可以着手进行调查。尤其是一些错误可能会导致已运行很长时间的作业中止时，这个技巧特别有用。

4.12　小结

本章介绍了如何开发MapReduce作业，重点讲述了可能经常会碰到的一些问题及其解决方法。特别是，我们学习了如何使用Hadoop Streaming脚本语言编写map和reduce任务，以及如何有效使用Streaming技术进行早期的作业原型设计和最初的数据分析。

我们同时也了解到，使用脚本语言编写任务有利于在命令行下直接测试和调试代码。我们还学习了Java API，使用`ChainMapper`类可以把复杂的map任务分解成一系列较小且更有针对性的map任务。

紧接着，我们学习了Distributed Cache如何在所有节点间共享数据。它将数据文件从HDFS拷贝到每个节点的本地文件系统，使我们可以在本地访问这些数据。我们还学到，通过定义Java枚举可以为计数器组添加计数器并利用Hadoop框架自动维护计数器值。此外，我们也了解了如何组合运用计数器、任务状态信息和debug日志进行高效的作业分析。

在开发MapReduce作业的过程中，你会经常碰到大部分前述技术和方法。下一章，我们将探索一系列更高级的且不会经常碰到的技术，但它们却十分重要。

第5章

高级MapReduce技术

既然我们已经学习了一些MapReduce基础知识及其使用方法，下面将研究更多MapReduce相关技术和概念。

本章包括以下内容：

- 对数据执行联结操作；
- 用MapReduce实现图算法；
- 如何用与语言无关的方式表示复杂数据类型。

同时，在研究学习案例的过程中，我们将强调一些实用的技巧、窍门和某些领域的最佳实践等内容。

5.1 初级、高级还是中级

本章标题中出现的“高级”一词有点危险，因为复杂性是一个主观概念。因此，我们需要界定清楚“高级”一词涵盖的内容。我们从不认为本章要讲的内容是需要花费多年时间才能领悟到的大智慧。反之，我们也不认为本章讨论的一些技术和问题适合Hadoop新手。

因此，本章使用“高级”一词指代最初几天或几周不曾学到或遇到的技术，或者即便遇到过却没有完全理解的内容。这些技术不仅提供了特定问题的专门解决方案，也强调了如何使用标准Hadoop与相关API解决明显不适用于MapReduce处理模型的问题。之后，我们也会提到一些替代方法，虽然本章不会具体学习如何实现这些方法，但它们可能有助于读者更深入地研究MapReduce。

我们要学习的第一个案例就属于后一种情况：在MapReduce作业中执行各种联结操作。

5.2 多数据源联结

没有问题会使用一个单独的数据集。在许多情况下，有一些简单的方法可以避免在

MapReduce框架中处理多个单独却又相关的数据集。

当然，本章提到的联结类似于关系数据库中的概念。在关系数据库中，将数据分成多个表，然后使用SQL联结语句从多个数据源获取数据是很自然的事。一个典型例子是，主表仅包含特定资料的唯一ID，可以使用与其他表的联结操作获取该ID指代的数据。

5.2.1　不适合执行联结操作的情况

在MapReduce中是可以实现联结操作的。实际上，稍后我们会看到，问题并不在于是否具备实现该功能的能力，而在于从众多潜在策略中选择哪一个。

然而，MapReduce联结通常较难编写，而且效率低下。无论你用了多长时间Hadoop，都会遇到需要进行联结操作的场景。不过，如果需要在MapReduce作业中频繁执行联结，你可能就要问一下自己，数据的结构性是不是很强以及是不是比预想的存在更多内在关联。如果是这样的话，可能你需要考虑使用**Apache Hive**（第8章的主要话题）或**Apache Pig**（在第8章会简要提到）。二者都提供了基于Hadoop的附加层，允许使用高级语言进行数据处理操作，比如Hive使用的就是一种SQL语言的变种。

5.2.2　map端联结与reduce端联结的对比

在Hadoop中，有两种基本方法可以实现数据联结。我们根据数据联结发生在作业执行的哪个阶段，将它们分别命名为map端联结和reduce端联结。无论哪种情况，都需要汇集多个数据流并通过一些逻辑执行数据联结。这两种方法的基本区别在于，多个数据流整合是发生在mapper函数还是reducer函数。

顾名思义，map端联结把数据流读入mapper，并利用编码在mapper函数内部的逻辑执行联结操作。map端联结的巨大优势在于，通过在mapper中执行全联结（更关键的是减少数据量），传输给reduce阶段的数据量大幅下降。map端联结也有缺点：要么得确保其中一个数据源的数据规模很小，要么需要按照特定规则定义作业的输入。通常，唯一的方法是使用另一个MapReduce作业对数据进行预处理，而该作业的唯一目的就是为map端联结准备数据。

与之相反，reduce端联结不在map阶段对多数据流执行联结操作，而是把联结操作放在reduce阶段进行。这种方法的潜在缺陷是，所有数据源的全部数据都经过shuffle阶段传入reducer，而其中大部分数据可能被联结操作丢弃。对大数据集而言，这将是一个明显开销。

reduce端联结的主要优点是简单。开发者基本上可以决定作业结构，而且通常为相关数据集定义reduce端联结的方法是很简单的。下面看一个例子。

5.2.3 匹配账户与销售信息

很多公司的销售记录与客户信息都是分开管理的。当然，二者之间存在某种关联：通常销售记录包含客户账户的唯一ID，而客户账户则与某些销售记录相关。

在Hadoop中，这些内容以两类数据文件表示：一类文件包含用户ID记录和销售信息，另一类文件包含每个客户账户的全部数据。

最常见的任务就是使用这些数据源生成报告。举个例子，我们想了解销售总额和针对每个客户的销售额，但我们更愿意看到销售额与客户姓名的对应关系，而非匿名的ID号。这一点非常重要，当客户服务代表想致电最活跃的顾客（通过分析销售记录得出活跃客户）时，他想要的是客户姓名，不是数字。

5.3 实践环节：使用 MultipleInputs 实现 reduce 端联结

我们可以通过执行reduce端联结完成上节提到的客户–销售额报告。执行以下步骤。

(1) 把下列以tab为分隔符的内容保存为`sales.txt`文件。

```
00135.992012-03-15
00212.492004-07-02
00413.422005-12-20
003499.992010-12-20
00178.952012-04-02
00221.992006-11-30
00293.452008-09-10
0019.992012-05-17
```

(2) 把下列以tab为分隔符的内容保存为`accounts.txt`。

```
001John AllenStandard2012-03-15
002Abigail SmithPremium2004-07-13
003April StevensStandard2010-12-20
004Nasser HafezPremium2001-04-23
```

(3) 把上述两个数据文件拷贝至HDFS。

```
$ hadoop fs -mkdir sales
$ hadoop fs -put sales.txt sales/sales.txt
$ hadoop fs -mkdir accounts
$ hadoop fs -put accounts/accounts.txt
```

(4) 把以下代码保存为`ReduceJoin.java`文件。

```
import java.io.* ;
```

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.*;
import org.apache.hadoop.mapreduce.lib.input.*;

public class ReduceJoin
{

    public static class SalesRecordMapper
extends Mapper<Object, Text, Text, Text>
{

        public void map(Object key, Text value, Context context)
throws IOException, InterruptedException
        {
            String record = value.toString() ;
            String[] parts = record.split("\t") ;

            context.write(new Text(parts[0]), new
Text("sales\t"+parts[1])) ;
        }
    }

    public static class AccountRecordMapper
extends Mapper<Object, Text, Text, Text>
{
        public void map(Object key, Text value, Context context)
throws IOException, InterruptedException
        {
            String record = value.toString() ;
            String[] parts = record.split("\t") ;

            context.write(new Text(parts[0]), new
Text("accounts\t"+parts[1])) ;
        }
    }

    public static class ReduceJoinReducer
    extends Reducer<Text, Text, Text, Text>
    {
        public void reduce(Text key, Iterable<Text> values,
            Context context)
            throws IOException, InterruptedException
        {
            String name = "" ;
double total = 0.0 ;
            int count = 0 ;

            for(Text t: values)
            {
                String parts[] = t.toString().split("\t") ;
```

```
                if (parts[0].equals("sales"))
                {
                    count++ ;
                    total+= Float.parseFloat(parts[1]) ;
                }
                else if (parts[0].equals("accounts"))
                {
                    name = parts[1] ;
                }
            }

            String str = String.format("%d\t%f", count, total) ;
            context.write(new Text(name), new Text(str)) ;
        }
    }
    public static void main(String[] args) throws Exception
{
        Configuration conf = new Configuration();
        Job job = new Job(conf, "Reduce端 join");
        job.setJarByClass(ReduceJoin.class);
        job.setReducerClass(ReduceJoinReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
MultipleInputs.addInputPath(job, new Path(args[0]),
TextInputFormat.class, SalesRecordMapper.class) ;
MultipleInputs.addInputPath(job, new Path(args[1]),
TextInputFormat.class, AccountRecordMapper.class) ;
        Path outputPath = new Path(args[2]);
        FileOutputFormat.setOutputPath(job, outputPath);
outputPath.getFileSystem(conf).delete(outputPath);

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

(5) 编译ReduceJoin.java文件并把它增加到join.jar文件中。

```
$ javac ReduceJoin.java
$ jar -cvf join.jar *.class
```

(6) 通过执行下列命令行运行作业。

```
$ hadoop jar join.jarReduceJoin sales accounts outputs
```

(7) 检查结果文件。

```
$ hadoop fs -cat /user/garry/outputs/part-r-00000
John Allen    3        124.929998
Abigail Smith 3        127.929996
April Stevens 1        499.989990
Nasser Hafez  1        13.420000
```

原理分析

首先，我们创建了本例用到的数据文件。之所以创建了两个小数据集，是因为这样更易于跟踪输出结果。我们定义的第一个数据集是账户详情，它由4个字段组成。

- 账户ID
- 客户姓名
- 账户类型
- 开户日期

接着，我们创建了销售记录数据集，它由如下3个字段组成。

- 购买者账户ID
- 销售额
- 售出时间

当然，实际的账户信息和销售记录的字段要比这里提到的多很多。创建了这些文件后，我们将其放到HDFS。

然后，我们创建了`ReduceJoin.java`文件，它看起来很像我们之前用过的MapReduce作业。这个ReduceJoin作业的某些内容使它变得与众不同，并实现了一个联结操作。

首先，ReduceJoin类定义了两个mapper。我们之前曾讲过，MapReduce作业可以包含多个链式执行的mapper。但本例中，我们希望为每个输入数据源都实现不同的mapper。于是，我们分别在`SalesRecorkMapper`和`AccountRecordMapper`类中定义了销售数据和账户数据。我们用到了`org.apache.hadoop.mapreduce.lib.io`包里的`MultipleInputs`类，如下所示。

```
MultipleInputs.addInputPath(job, new Path(args[0]),
TextInputFormat.class, SalesRecordMapper.class) ;
MultipleInputs.addInputPath(job, new Path(args[1]),
TextInputFormat.class, AccountRecordMapper.class) ;
```

如你所见，之前的例子中只有一个输入数据源，而本例则与之不同，`MultipleInputs`类允许我们有多个数据源，并为每个数据源指定独立的输入格式和mapper。

mapper的实现相当简单，`SalesRecordMapper`类的输出格式为`<account number>, <sales value>`，而`AccountRecordMapper`类的输出格式为`<account number>, <client name>`。因此，我们把排序后的各次销售额和客户姓名传入reducer，在reducer中执行数据联结。

请注意，实际上两个mapper都输出了多余数据。`SalesRecordMapper`类在输出销售额时以`sales`为前缀，而`AccountRecordMapper`类以`account`为前缀。

如果我们看一下reducer，就会明白这样做的原因。reducer会获取给定键的每条记录，但如果

不使用这些明确的标签，我们就不知道给定值是sales mapper的输出还是account mapper的输出，因此也就不知道该怎样处理这些数据。

因此，ReduceJoinReducer类根据Iterator对象中的值来自哪个mapper，决定对其采取相应的处理措施。AccountRecordMapper类的输出（应该有且仅有一个）被用于生成作业最终输出结果中的客户姓名。针对每位客户的销售记录（很可能是多个的，因为大部分客户会买多样商品）被用于计算订单总数和总消费额。因此，reducer输出数据的键是账户持有人姓名，值是包含订单总数和总消费额的字符串。

我们编译并执行ReduceJoin类。请注意，我们输入了三个参数，其中两个参数表示输入路径，另一个参数表示输出路径。根据MultipleInputs类的不同配置方式，我们必须保证以正确的顺序输入三个参数。MapReduce作业中不存在判定哪种类型的文件在哪个目录下的动态机制。

执行完毕后，我们检查输出文件并确认输出文件包含了客户姓名及其总消费额。

DataJoinMapper和TaggedMapperOutput

还有一种实现reduce端联结的方法，这种方法更为复杂，而且是面向对象的。DataJoin-MapperBase 和TaggedMapOutput类在org.apache.hadoop.contrib.join包里，它们封装了获取map输出标签并将它们传给reducer的方法。也就是说，有了这种方法，我们不必再像刚才那样显式定义标签字符串，然后小心解析reducer收到的数据，才能识别数据是哪个mapper输出的。Hadoop提供的DataJoinMapperBase 和TaggedMapOutput类已经封装了实现该功能的函数。

尤其是处理数值型或非文本型数据时，这个功能特别重要。因为如果像之前例子那样创建明确的标签的话，我们必须将整型数据转换成字符串，然后才能添加所要求的标签前缀。与使用标准格式的数值类型并使用其他类来实现标签的方法相比，上述方法的效率要低得多。

Hadoop框架支持复杂的标签生成以及标签分组，我们之前没有实现这些功能。要使用这些功能，还需要完成一些其他工作，包括重写某些方法以及使用另外的map基类。对于像上例这么简单的联结来讲，使用Hadoop框架提供的标签处理机制有点大材小用，但如果要实现非常复杂的标签处理代码，值得试试它。

5.3.1　实现map端联结

要想在某个阶段实现数据联结，必须能够在相应的时间访问每个数据集的适用记录。这就是reduce端联结易于实现的原因。尽管reduce端联结会消耗额外的网络流量，但显然，reducer掌握所有与联结键相关的记录。

如果我们想在mapper中执行联结，上述条件并不容易满足。我们不能保证输入数据的结构化

程度足够好，可以同时读出关联记录。这里我们大致有两种方法：避免从多个外部源读取数据或预处理数据使其可用于map端联结。

1. 使用Distributed Cache

实现第一种方法的最简单方式就是，只使用一个数据集并把它放入上一章讲到的Distributed Cache。该方法可用于多个数据源，但简单起见，我们这里只讨论两个数据源的情况。

如果我们有一个较大数据集和一个较小数据集，例如之前的销售记录和账户信息，一种做法是把账户信息打包放入Distributed Cache。然后每个mapper将这些数据读入一个高效的数据结构，如使用联结键作为哈希键的哈希表。接着处理销售记录，在处理过程中，可以从哈希表中获取要用的每条账户信息。

这种方法非常有效，尤其是当较小的数据集可以轻易读入内存时，这是一种很好的做法。然而，我们并非总是如此幸运，有时最小数据集的规模也很大，以至于无法将其拷贝到每台工作机并保存在内存里。

一展身手：实现map端联结

以刚才的销售记录/账户记录为例，使用Distributed Cache实现map端联结。如果将账户记录载入哈希表中实现账户ID和客户名字的映射，那么就可以用账户ID获取客户姓名。这样就可以在mapper处理销售信息时实现map端联结。

2. 裁剪数据以适合缓存大小

如果最小数据集对Distributed Cache来说还是太大，并不需要舍弃所有字段。比如，在前面的例子中，我们仅从每条记录中提取了两个字段并丢弃了作业不需要的其他字段。实际上，账户有多个属性，这种减少数据量的方法会显著降低数据规模。通常，Hadoop可访问的数据是完整数据集，但是我们需要的只是其中若干字段。

因此，在此情况下，我们可以从完整数据集中提取MapReduce作业需要的字段，通过这种方式创建的裁剪数据集的大小完全适合在缓存中使用。



这种方法与**面向列的数据库**（column-oriented databases）的概念非常相似。传统的关系数据库每次存储一行数据，这就意味着需要读取完整的一行，然后从中提取某一列的内容。基于列的数据库则是分别存储每一列，允许每次查询直接读取用户感兴趣的列。

如果采用这种方法，需要考虑何种机制可以被用来生成数据子集，以及执行这些操作的频率。

一种直接的方法是编写另外一个MapReduce作业，用它完成必需的过滤，在后续的作业中将该作业的输出用在Distributed Cache中。如果较小数据集的变动很小，就避免了按照预定计划生成裁剪数据集的麻烦。例如，每晚刷新数据集。否则，你需要用到由两个MapReduce作业组成的作业链：一个作业用于生成裁剪数据集，另一个作业利用原始数据集和Distributed Cache中的数据执行联结操作。

3. 使用代表数据而非原始数据

有时，我们使用某个数据源并非是为了获取额外数据，而是为了推导出一些辅助决策的结论。例如，我们可能希望通过分析销售记录，从中提取那些配送地址属于某一特定区域的记录。

在这种情况下，我们可以将所需数据的规模降至一列合适的销售记录，它们更易被放入缓存。同样，我们可以将其存入只存储有效记录的哈希表，甚至使用类似排序列表或排序树的形式。在可以接受误报却须保证没有漏报的情况下，可以使用**Bloom filter**紧凑地表示此类信息。

可以看出，采用这种方法实现map端联结需要创新，并与数据集的性质和手头的问题有很大关系。但是请记住，最好的关系型数据库管理员耗费大量时间剔除非必要的数据处理，以达到优化查询的目的。因此，读者需要时常问问自己是否真的需要处理所有数据。

5

4. 使用多个mapper

从根本上说，上述技术都在尽力避免实现多个完整数据集之间的联结。但有时不得不这样做，因为你可能会遇到非常庞大的数据集，而且无法用这些方法合并。

`org.apache.hadoop.mapreduce.lib.join`包里有一些类支持这种需求。`CompositeInputFormat`是一个有趣的主类，它使用用户定义的函数合并多个数据源的记录。

这种方法的主要缺点是，所有数据源必须以相同键为索引，并且以相同方法进行排序和分块。原因很简单，从每个数据源读取数据时，Hadoop框架需要知道某个特定键是否出现在每条记录中。假如每个分块都是有序的，并包括相同的键，可以使用简单的迭代程序完成所需的匹配。

显然，要处理的数据不可能恰好满足上述条件，所以读者需要自己编写预处理作业，将所有的输入数据源转化成正确的顺序和分块结构。



本节讨论的内容开始涉及分布式联结和并行联结算法，这些课题都经历了广泛的学术研究和商业研究。如果你对这些内容感兴趣并想学习更多的基础理论，到http://scholar.google.com搜索相关内容。

5.3.2　是否进行联结

结束了MapReduce的联结之旅后，让我们回到最开头的问题：你是否真要进行联结操作？读者常常需要在较易实现却效率不高的reduce端联结和较为高效也更为复杂的map端联结之间做出选择。我们已看到，的确可在MapReduce中实现多个数据源的联结，但有时源数据并不适合进行联结操作。如果执行联结操作需要用户投入大量精力，我们推荐使用Hive或Pig。显然，我们可以使用上述工具，它们在后台生成MapReduce代码，并直接实现map端和reduce端联结。但最好使用一个精心设计、经过优化的代码库完成这些工作，而不要自行实现。这也正是我们要使用Hadoop而不自己编写分布式处理框架的原因。

5.4　图算法

所有杰出的计算机科学家都认为图数据结构是最强大的工具之一。很多复杂系统都由图来表示，至少几十年前就出现了强大的算法知识体系来解决大量图问题。但由于其本身性质，图及图算法通常很难用MapReduce范式描述。

5.4.1　Graph 101

退一步，我们先对图的相关术语进行定义。图是由多个通过边相连的节点（也被称为顶点）组成的数据结构。根据图的类型，边可以是单向边也可以是双向边，也可以有与之关联的权重。例如，一个城市的路网可以看做一张图，其中路是图的边，路的交点和感兴趣的点是图的节点。有些路是单向路，有些不是；有些路要收通行费，有些路在某些时候处于封闭状态，等等。

对物流公司而言，选择一条从一个地方到另一个地方的最佳路径可以节省很多钱。不同的图算法通过综合考虑各条路的属性得出最佳路径。这些属性包括是否是单行道，以及用权重表示的通行成本，它决定了特定道路是否更具吸引力。

一个更时髦的例子是社交关系图，随着Facebook之类的网站越来越受欢迎，这种社交关系图也逐渐普及起来。社交关系图视可被为以人为节点，以他们之间的关系为边的图。

5.4.2　图和MapReduce

图与其他MapReduce问题的主要区别在于，图处理是有状态的，这可以从节点间的路径关系和图算法处理的大量节点间的路径关系看出来。图算法倾向于根据全局状态决定下一步要处理的节点，并在每步操作中修改全局状态。

尤其是，大多数著名的算法通常以增量方式或可重入方式执行，它们用不同的数据结构表示

已处理节点和待处理节点，然后对未处理节点进行运算并将该节点加入到已处理节点集合中。

但是，从概念上讲，MapReduce作业是无状态的。它基于分而治之的方法，每台Hadoop主机处理全部数据的一小部分，并输出作业最终结果的一部分，而整个作业的最终输出可以视为这些子任务输出的聚合。因此，使用Hadoop实现图算法时，我们需要把原本有状态的单线程算法用无状态的、并行的分布式框架来描述。这就是最大的挑战。

大部分著名的图算法为了找出节点间符合需求的路由（通常以某种成本为衡量标准），通常使用图搜索或图遍历的办法。最基础的图遍历算法是**DFS**（depth-first search，深度搜索算法）和**BFS**（breadth-first search，广度搜索算法）。两者区别在于，某一节点及其邻居节点被处理的先后顺序不同。

接下来，我们会看到一种特殊的遍历算法，它为图中的给定起始节点计算图中其余所有节点与该节点的距离。

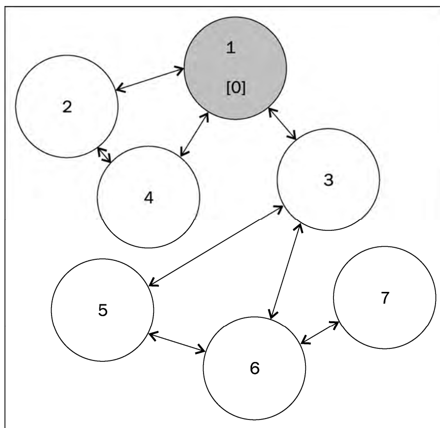


可以看出，图论和图算法是一个巨大的研究领域，本书对其介绍非常有限。如果读者想了解更多内容，可以从Wikipedia中关于图的条目开始，参见http://en.wikipedia.org/wiki/Graph_(abstract_data_type)。

5.4.3 图的表示方法

我们面临的第一个问题就是，如何表示图才能使用MapReduce对其进行高效处理。有几种图的表示方法为大家所熟知，例如基于指针的表示法，邻接矩阵表示法和邻接表表示法。在大多数实现中，这些表示法通常假设整个图可以存储在单个进程空间。我们需要对这些表示法进行修改，允许独立的map和reduce任务对个别节点进行处理。

在后面的例子中，我们以下图为例进行介绍。该图还包含一些稍后会讲到的额外信息。



这个图很简单，只有7个节点，其中只有一条单向边，其余都是双向边。我们使用了标准图算法使用的着色方法，其中：

- 白色表示未处理节点；
- 灰色表示正在处理的节点；
- 黑色表示已处理节点。

随着我们在下列操作中逐步处理各个节点，节点状态会按照上述约定发生变化。

5.5　实践环节：图的表示

首先定义一种图的文本表示法，后面的例子中会用到该方法。

新建graph.txt文件，其内容如下：

```
12,3,40C
21,4
31,5,6
41,2
53,6
63,5
76
```

原理分析

我们定义了表示图的文件结构，在某种程度上，它借鉴了图的邻接表表示法。假设每个节点有1个唯一的ID，该文件结构包括4个字段，具体内容如下：

- 节点ID
- 以逗号分隔的邻居列表
- 与起始点的距离
- 节点状态

最初，只有起始节点的第3个和第4个字段有值：它与自身的距离为0，状态为“C”。具体原因会在后续内容进行解释。

这个图是一个有向图，也就是说，节点1指向节点2的路径与节点2指向节点1的路径是两条不同的路径。从图示中可以看到，除一条边外，其余边的两个端点都有箭头。

算法综述

由于该算法及相应的MapReduce作业相当复杂，我们将先解释算法，然后展示代码，最后在

算法的使用过程中说明其原理。

基于上述表示方法，我们定义一个可多次执行的MapReduce作业，以获得最终输出。每次执行作业时，使用上次作业运行的输出作为本次运行的输入。

基于上节描述的色码，我们定义了节点的三种状态，如下所述。

- Pending：节点尚未被处理。这是节点的默认状态，对应白色节点。
- Currently processing：节点正在被处理，对应灰色节点。
- Done：已算出了节点与起始点的最终距离，对应黑色节点。

1. mapper

mapper读取图的当前状态，并按照下述方法处理节点。

- 如果节点状态为Done，原封不动将其输出。
- 如果节点状态为Currently processing，把它的状态改为Done，然后输出。它的邻居节点的输出与该节点类似，只是在其距离值的基础上加1，而邻居节点保持不变。例如，节点1不知道节点2的邻居节点。
- 如果节点状态为Pending，将其状态改为Currently processing并输出。

2. reducer

reducer会收到每个节点的一条或多条记录，它会把这些记录值合并成节点的最终输出。

reducer的主要算法如下：

- 状态为Done的节点的输出即为最终输出，无需对其值进一步处理；
- 对处于其他状态的节点，提取其邻居列表，从中找出最远距离和节点状态作为最终输出。

3. 迭代应用

如果我们应用一次该算法，节点1被标记为Done，它的直接邻居被标记为Currently processing，其余节点都被标记为Pending。依次应用该算法会使所有节点都转为Done状态。因为每个节点总会被处理，它的邻居都被纳入正在处理队列。稍后我们会演示该过程。

5.6　实践环节：创建源代码

我们现在讲解实现图遍历算法的源代码。因为代码较长，我们将其分为几个部分。显然，这些代码应当放在同一个源文件中。

(1) 新建GraphPath.java文件，添加下列引用声明。

```
import java.io.* ;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.*;
import org.apache.hadoop.mapreduce.lib.output.*;

public class GraphPath
{
```

(2) 创建内部类，以面向对象的方法表示节点。

```
// Inner class to represent a node
    public static class Node
    {
// The integer node id
        private String id ;
// The ids of all nodes this node has a path to
        private String neighbours ;
// The distance of this node to the starting node
        private int distance ;
// The current node state
        private String state ;

// Parse the text file representation into a Node object
        Node( Text t)
        {
            String[] parts = t.toString().split("\t") ;
this.id = parts[0] ;
this.neighbours = parts[1] ;
            if (parts.length<3 || parts[2].equals(""))
this.distance = -1 ;
            else
this.distance = Integer.parseInt(parts[2]) ;

            if (parts.length< 4 || parts[3].equals(""))
this.state = "P" ;
            else
this.state = parts[3] ;
        }

// Create a node from a key and value object pair
        Node(Text key, Text value)
        {
            this(new Text(key.toString()+"\t"+value.toString())) ;
        }

        Public String getId()
        {return this.id ;
        }
```

```
        public String getNeighbours()
        {
            return this.neighbours ;
        }

        public int getDistance()
        {
            return this.distance ;
        }

        public String getState()
        {
            return this.state ;
        }
    }
```

(3) 创建作业的mapper。该mapper新建一个Node对象接收输入数据，并根据Node对象的状态执行相应操作。

```
    public static class GraphPathMapper
extends Mapper<Object, Text, Text, Text>
{

       public void map(Object key, Text value, Context context)
throws IOException, InterruptedException
{
         Node n = new Node(value) ;

         if (n.getState().equals("C"))
         {
//    Output the node with its state changed to Done
           context.write(new Text(n.getId()), new
Text(n.getNeighbours()+"\t"+n.getDistance()+"\t"+"D")) ;

                for (String neighbour:n.getNeighbours().
split(","))
                {

// Output each neighbour as a Currently processing node
// Increment the distance by 1; it is one link further away
                    context.write(new Text(neighbour), new
Text("\t"+(n.getDistance()+1)+"\tC")) ;
                }
            }
            else
            {
// Output a pending node unchanged
                context.write(new Text(n.getId()), new
Text(n.getNeighbours()+"\t"+n.getDistance()
+"\t"+n.getState())) ;
            }

        }
    }
```

5

(4) 创建作业的reducer。与mapper一样，该reducer读取节点记录，并根据节点状态输出不同的值。基本方法是，从输入数据提取状态和距离字段的最大值，并把这些值汇聚成最终输出。

```
    public static class GraphPathReducer
extends Reducer<Text, Text, Text, Text>
{

        public void reduce(Text key, Iterable<Text> values,
            Context context)
            throws IOException, InterruptedException
{
// Set some default values for the final output
            String neighbours = null ;
int distance = -1 ;
String state = "P" ;

            for(Text t: values)
{
                Node n = new Node(key, t) ;
                if (n.getState().equals("D"))
                {
// A done node should be the final output; ignore the remaining
// values
neighbours = n.getNeighbours() ;
                    distance = n.getDistance() ;
                    state = n.getState() ;
                    break ;
                }

// Select the list of neighbours when found
                if (n.getNeighbours() != null)
neighbours = n.getNeighbours() ;

// Select the largest distance
                if (n.getDistance() > distance)
distance = n.getDistance() ;

// Select the highest remaining state
                if (n.getState().equals("D") ||
(n.getState().equals("C") &&state.equals("P")))
state=n.getState() ;
            }

// Output a new node representation from the collected parts
            context.write(key, new
Text(neighbours+"\t"+distance+"\t"+state)) ;
        }
    }
```

(5) 创建作业驱动。

```
    public static void main(String[] args) throws Exception
{
```

```
            Configuration conf = new Configuration();
            Job job = new Job(conf, "graph path");
            job.setJarByClass(GraphPath.class);
            job.setMapperClass(GraphPathMapper.class);
            job.setReducerClass(GraphPathReducer.class);
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(Text.class);
            FileInputFormat.addInputPath(job, new Path(args[0]));
            FileOutputFormat.setOutputPath(job, new Path(args[1]));
            System.exit(job.waitForCompletion(true) ? 0 : 1);
        }
    }
```

原理分析

该作业实现了之前描述的算法，下节将会执行该算法。作业设置没什么特别的，除了算法定义，还首次使用内部类来表示节点。

通常，mapper或reducer的输入是对复杂结构或对象的扁平化表示。我们可以使用那种表示方法，但这样会导致mapper和reducer中充满了文本和字符串操作代码，不利于理解算法本身。

使用Node内部类，实现了从扁平的文本文件向封装对象的映射，这种用对象表示节点的方法在业务领域很有意义。从语义来说，不同对象之间的属性对比要比字符串对比更有意义，这也使得mapper和reducer的逻辑更为清楚。

5.7 实践环节：第一次运行作业

现在，我们使用该算法对图进行首次运算。

(1) 将之前创建的graph.txt文件放到HDFS。

```
$ hadoop fs -mkdirgraphin
$ hadoop fs -put graph.txtgraphin/graph.txt
```

(2) 编译源文件并创建JAR文件。

```
$ javac GraphPath.java
$ jar -cvf graph.jar *.class
```

(3) 执行MapReduce作业。

```
$ hadoop jar graph.jarGraphPathgraphingraphout1
```

(4) 检查输出文件。

```
$ hadoop fs -cat /home/user/hadoop/graphout1/part-r00000
12,3,40D
```

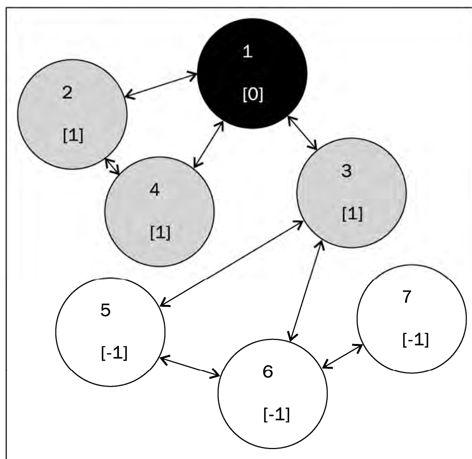
```
21,41C
31,5,61C
41,21C
53,6-1P
63,5-1P
76-1P
```

原理分析

将数据源文件放到HDFS上并创建了作业JAR文件之后，我们在Hadoop上执行作业。输出结果显示，该图发生了一些变化，具体如下：

- 节点1被标记为Done状态。显然，它与自身的距离是0；
- 节点2、3、4作为节点1的邻居，被标记为Currently processing节点；
- 其余节点都被标记为Pending。

经过一次MapReduce处理之后，该图状态如下所示。



运行结果和算法描述相吻合，一切都在预料之中。第一个节点已处理完毕，mapper提取的邻居节点正在处理中，其余节点还没开始处理。

5.8　实践环节：第二次运行作业

假如我们把上步的输出结果作为下次作业运行的输入，我们认为节点2、3、4将处于Done（已处理）状态，它们的邻居节点将处于Currently processing（正在处理）状态。执行下列步骤，看看我们的猜想是否正确。

(1) 执行下列命令，运行MapReduce作业。

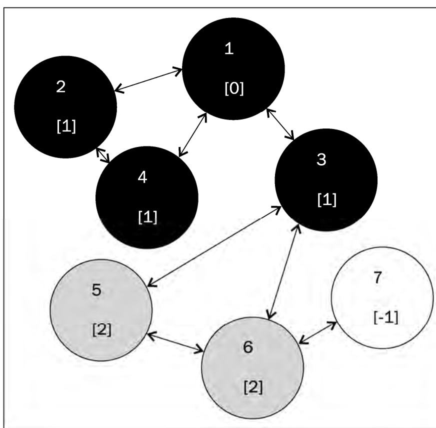
```
$ hadoop jar graph.jarGraphPathgraphout1graphout2
```

(2) 检查输出文件。

```
$ hadoop fs -cat /home/user/hadoop/graphout2/part-r000000
12,3,40D
21,41D
31,5,61D
41,21D
53,62C
63,52C
76-1P
```

原理分析

果然不出所料，经过第二次处理后，节点1~4处于已处理状态，节点5和节点6处于正在处理状态，节点7处于未处理状态，如下图所示。



假如再次运行作业，我们认为，节点5和节点6的状态会变为Done，其余未处理的邻居节点状态会变为Currently processing。

5.9 实践环节：第三次运行作业

第三次执行算法，验证我们的预测是否正确。

(1) 执行MapReduce作业。

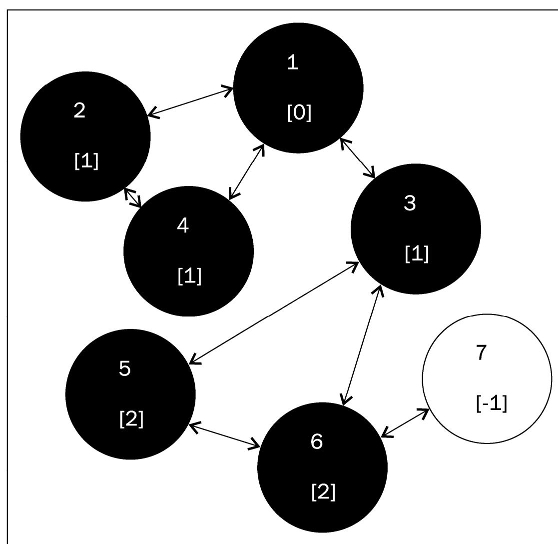
```
$ hadoop jar graph.jarGraphPathgraphout2graphout3
```

(2) 检查输出文件。

```
$ hadoop fs -cat /user/hadoop/graphout3/part-r-00000
12,3,40D
21,41D
31,5,61D
41,21D
53,62D
63,52D
76-1P
```

原理分析

现在，我们看到节点1~6处于Done状态。但节点7仍处于Pending（未处理）状态，且没有节点正在被处理，如下图所示。



之所以出现这种情况，是由于虽然存在一条以节点7为起点、节点6为终点的边，却不存在反向边，造成节点6无法访问节点7。因此，从节点1出发无法到达节点7。如果最后运行一次算法，我们认为该图保持不变。

5.10　实践环节：第四次也是最后一次运行作业

让我们再运行一次作业，验证作业输出是否已达到最终稳定状态。

(1) 执行MapReduce作业。

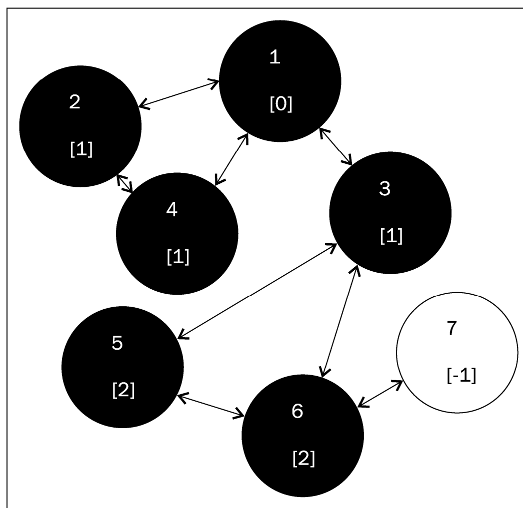
```
$ hadoop jar graph.jarGraphPathgraphout3graphout4
```

(2) 检查输出文件。

```
$ hadoop fs -cat /user/hadoop/graphout4/part-r-00000
12,3,40D
21,41D
31,5,61D
41,21D
53,62D
63,52D
76-1P
```

原理分析

输出的结果与预期一致。因为从节点1或其邻居节点出发无法到达节点7，所以节点7的状态仍为Pending（未处理），且永远无法对其进行处理。因此，该图已达到稳定状态，其输出保持不变，如下图所示。



我们的算法中未加入对其是否满足结束条件的判断。如果经过某次运算后，该图不再产生新的Done或Currently processing节点，即完成了整个运算过程。

本章，我们使用手工方法判断算法是否满足结束条件，也就是说，我们通过实验得知该图已达到最终的稳定状态。然而，可以通过编程实现这个功能。在下一章中，我们会学习自定义的作业计数器，使用它即可实现该功能。例如，每当新建一个Done或Currently processing节点的时候，计数器的值加1。只有在某次作业完成运行后，计数器的值大于0时，才会再次执行作业。否则，即可认为该图达到了最终稳定状态。

5.10.1 运行多个作业

在上述算法中，我们首次明确地把一个MapReduce作业的输出当做另一个作业的输入。在大多数情况下，这些接续执行的作业之间是有所区别的。然而，正如我们所看到的，反复执行同一个算法直到其输出达到稳定状态是很有意义的。

5.10.2 关于图的终极思考

对熟悉图算法的人来说，上述处理方法显得非常陌生。其实，这是使用一系列无状态的MapReduce作业实现一个有状态的、递归的、可重入的图算法的必然结果。我们要强调的并非用到的特定算法，而是如何采用纯文本结构和一系列MapReduce作业实现图遍历的经验。你可能会遇到一些乍看上去似乎无法使用MapReduce范式解决的问题。此时，认真思考本章用到的一些技术，同时请记住，许多算法都可以使用MapReduce建模。它们可能看似与传统方法有较大的差别，但我们的目标是输出正确结果，而非实现一种已知算法。

5.11 使用语言无关的数据结构

开发者往往会批评Hadoop的一切设计都以Java为中心，Hadoop团队一直在努力解决这个问题。这种批评似乎有点奇怪，难到一个用Java语言实现的项目不该以Java为中心吗？但从用户的角度考虑，他们希望不依赖于特定语言实现对Hadoop的操作。

在第4章，我们已演示了使用Hadoop Streaming脚本实现map和reduce任务的方法。同时，Pipes也提供了类似的使用C++实现map和reduce任务的技术。然而，有一个地方仍只能使用Java，它就是Hadoop MapReduce支持的输入格式。效率最高的输入格式是SequenceFile, 它是一种支持压缩和分块的二进制文件。但是，SequenceFile只提供了Java API，用户无法使用其他语言读写SequenceFile。

我们可以使用外部进程创建用于MapReduce作业的数据，最佳方式是用文本格式输出生成的数据，或者对生成的数据进行预处理，将其转换为SequenceFile。我们还要想办法简化复杂数据类型的表示：要么将它们转换为文本格式，要么编写两种二进制格式的转换器。这些方法都不是很理想。

5.11.1 候选技术

幸运的是，近年来出现了一些技术可以解决这种跨语言的数据表示问题。这些技术包括**Protocol Buffers**（Google创建的项目，其网址为http://code.google.com/p/protobuf）、**Thrift**（该项目最初由Facebook创建，目前是一个Apache子项目，其网址为http://thrift.apache.org）以及**Avro**（该项目由Hadoop的创建者Doug Cutting所创建）。由于Avro的创建者与Hadoop相同，我们将使用

它来研究这个问题。本书中不会讲到Thrift和Protocol Buffers，但它们都是成熟的技术。如果你对数据序列化的问题感兴趣，请访问它们的主页获取更多信息。

5.11.2 Avro简介

Avro是一种数据存储框架，可使用多种编程语言对其进行操作，其主页地址为http://avro.apache.org。Avro创建了一种可压缩和可切分的二进制结构化数据格式，换句话说，它可以有效地用于向MapReduce作业输入数据。

Avro支持层次化的数据结构，因此，我们可以在一个Avro记录中嵌套数组、枚举类型甚至是一个子记录。我们可以用任何程序语言创建此类数据文件，用Hadoop对其进行处理，并使用第三方语言读取结果。

下一节，我们将讨论Avro的语言独立性，而其表达复杂结构类型的能力也很重要。即便只使用Java语言，我们可以把用Avro表示的复杂数据结构作为mapper和reducer的输入输出类型。即使是像图节点这么复杂的数据结构都没问题。

5.12 实践环节：获取并安装 Avro

下面，我们将下载Avro并将其安装在自己的主机上。

(1) 从http://avro.apache.org/releases.html下载Avro的最新稳定版本。

(2) 从http://paranamer.codehaus.org下载最新版的ParaNamer库。

(3) 将下述3个JAR类添加到build classpath，以供Java编译器所用。

```
$ export CLASSPATH=avro-1.7.2.jar:${CLASSPATH}
$ export CLASSPATH=avro-mapred-1.7.2.jar:${CLASSPATH}
$ export CLASSPATH=paranamer-2.5.jar:${CLASSPATH}
```

(4) 将Hadoop安装路径下的JAR文件添加到`build` classpath。

```
Export CLASSPATH=${HADOOP_HOME}/lib/Jackson-core-asl-
1.8.jar:${CLASSPATH}
Export CLASSPATH=${HADOOP_HOME}/lib/Jackson-mapred-asl-
1.8.jar:${CLASSPATH}
Export CLASSPATH=${HADOOP_HOME}/lib/commons-cli-
1.2.jar:${CLASSPATH}
```

(5) 将新JAR文件添加到Hadoop `lib`目录。

```
$cpavro-1.7.2.jar ${HADOOP_HOME}/lib
$cpavro-1.7.2.jar ${HADOOP_HOME}/lib
$cpavro-mapred-1.7.2.jar ${HADOOP_HOME}/lib
```

原理分析

Avro的设置稍微有点复杂，它的创建时间比我们将用到的其他Apache工具晚得多，因此，它的设置不光是下载一个文件那么简单。

我们从Apache网站下载了avro-1.7.2.jar和avro-mapred-1.7.2.jar文件。因为这两个文件依赖于ParaNamer，所以我们又从http://paranamer.codehaus.org下载了`paranamer-2.5.jar`。



> 作者在创作本书时，ParaNamer主页上的下载链接出现了问题。作为替代方案，试一下下面这个下载链接：http://search.maven.org/remotecontent?filepath=com/thoughtworks/paranamer/paranamer/2.5/paranamer-2.5.jar

在下载到上述JAR文件之后，需要把它们添加到Java编译器要用到的classpath中。与此同时，我们还需要把编译和运行Avro代码要用到的几个包添加到Hadoop的`build` classpath。

最后，我们把这三个新JAR文件拷贝到集群中每台主机的Hadoop `lib`目录下。这样，在运行map和reduce任务的时候就可以使用这些类。我们也可以使用其他方法实现上述JAR文件的分发，但这是最简单的方式。

Avro及其模式

与Thrift和Protocol Buffers之类的工具相比，Avro的一个优势在于它对描述数据文件的模式的处理方法。其他工具都要求模式以独立资源的形式存在，而Avro数据文件将模式编码到该文件头部，这样，代码无需独立的模式文件即可解析数据文件。

Avro支持但不需要代码生成功能，该功能为特定的数据模式生成定制代码。在某些情况下，这是一种重要的优化措施，却不是必须的。

因此，我们可以写出一系列从来没有真正用到数据文件模式的Avro例程，但我们只会在部分数据处理任务中那样做。下面的例子中，我们定义一个表示删减过的UFO目击事件记录的数据模式。

5.13　实践环节：定义模式

下面，我们将在单独的Avro模式文件中创建简化的UFO模式。

将以下内容保存为`ufo.avsc`文件。

```
{ "type": "record",
  "name": "UFO_Sighting_Record",
  "fields" : [
    {"name": "sighting_date", "type": "string"},
```

```
    {"name": "city", "type": "string"},
    {"name": "shape", "type": ["null", "string"]},
    {"name": "duration", "type": "float"}
]
}
```

原理分析

可以看出，Avro在其模式中使用了JSON格式，Avro模式文件名通常以.avsc为后缀。刚才，我们创建了一个模式，其格式包括4个字段，各字段的含义如下所示：

- **Sighting_date**字段类型为string，它以yyyy-mm-dd这样的形式保存日期；
- **City**字段类型为string，它表示的是UFO目击事件发生的城市名；
- **Shape**字段类型为string，这是一个可选字段，表示的是UFO的形状；
- **Duration**字段以小数形式表示目击事件的持续时间，该字段以分钟为单位。

我们将按照已定义的模式，创建一些样本数据。

5

5.14 实践环节：使用 Ruby 创建 Avro 源数据

下面，我们使用Ruby创建样本数据，以说明Avro的跨语言特性。

(1) 安装rubygems包。

```
$ sudo apt-get install rubygems
```

(2) 安装Avro gem。

```
$ gem install avro
```

(3) 将下列代码保存为generate.rb文件。

```
require 'rubygems'
require 'avro'

file = File.open('sightings.avro', 'wb')
schema = Avro::Schema.parse(
File.open("ufo.avsc", "rb").read)

writer = Avro::IO::DatumWriter.new(schema)
dw = Avro::DataFile::Writer.new(file, writer, schema)
dw<< {"sighting_date" => "2012-01-12", "city" => "Boston", "shape"
=> "diamond", "duration" => 3.5}
dw<< {"sighting_date" => "2011-06-13", "city" => "London", "shape"
=> "light", "duration" => 13}
dw<< {"sighting_date" => "1999-12-31", "city" => "New York", "shape"
=> "light", "duration" => 0.25}
```

```
dw<< {"sighting_date" => "2001-08-23", "city" => "Las Vegas",
"shape" => "cylinder", "duration" => 1.2}
dw<< {"sighting_date" => "1975-11-09", "city" => "Miami",
"duration" => 5}
dw<< {"sighting_date" => "2003-02-27", "city" => "Paris", "shape"
=> "light", "duration" => 0.5}
dw<< {"sighting_date" => "2007-04-12", "city" => "Dallas", "shape"
=> "diamond", "duration" => 3.5}
dw<< {"sighting_date" => "2009-10-10", "city" => "Milan", "shape"
=> "formation", "duration" => 0}
dw<< {"sighting_date" => "2012-04-10", "city" => "Amsterdam",
"shape" => "blur", "duration" => 6}
dw<< {"sighting_date" => "2006-06-15", "city" => "Minneapolis",
"shape" => "saucer", "duration" => 0.25}
dw.close
```

(4) 运行generate.rb创建数据文件。

```
$ ruby generate.rb
```

原理分析

在使用Ruby之前，要保证已在Ubuntu主机上安装了rubygems包。随后，我们为Ruby安装它要用到的Avro gem，该文件早已存在。上述操作提供了使用Ruby语言读写Avro文件需要的代码库。

generate.rb脚本仅负责读入之前定义的模式，并生成一个包含10条测试记录的数据文件。接着，我们运行该脚本生成测试数据。

本节内容不会讲解Ruby的使用，因此读者需要自行分析程序中用到的Ruby API。相应的文档参见http://rubygems.org/gems/avro。

5.15 实践环节：使用 Java 语言编程操作 Avro 数据

既然上一节已经生成了Avro数据，接下来，编写Java程序对这些数据进行操作。

(1) 把下列代码保存为InputRead.java文件。

```
import java.io.File;
import java.io.IOException;

import org.apache.avro.file.DataFileReader;
import org.apache.avro.generic.GenericData;
import org.apache.avro. generic.GenericDatumReader;
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.io.DatumReader;

public class InputRead
{
    public static void main(String[] args) throws IOException
```

```
    {
        String filename = args[0] ;

        File file=new File(filename) ;
DatumReader<GenericRecord> reader= new
GenericDatumReader<GenericRecord>();
DataFileReader<GenericRecord>dataFileReader=new
DataFileReader<GenericRecord>(file,reader);

        while (dataFileReader.hasNext())
        {
GenericRecord result=dataFileReader.next();
            String output = String.format("%s %s %s %f",
result.get("sighting_date"), result.get("city"),
result.get("shape"), result.get("duration")) ;
System.out.println(output) ;
        }
    }
}
```

(2) 编译并运行`InputRead.java`。


```
$ javacInputRead.java
$ java InputReadsightings.avro
```

输出结果应当如下图所示。

```
hadoop@vm16: ~/avro
File Edit View Terminal Help
hadoop@vm16:~/avro$ javac InputRead.java
hadoop@vm16:~/avro$ java InputRead sightings.avro
2012-01-12 Boston diamond 3.500000
2011-06-13 London light 13.000000
1999-12-31 New York light 0.250000
2001-08-23 Las Vegas cylinder 1.200000
1975-11-09 Miami null 5.000000
2003-02-27 Paris light 0.500000
2007-04-12 Dallas diamond 3.500000
2009-10-10 Milan formation 0.000000
2012-04-10 Amsterdam blur 6.000000
2006-06-15 Minneapolis saucer 0.250000
hadoop@vm16:~/avro$
```

原理分析

我们定义了Java类InputRead，它通过命令行参数接收待处理的文件名，并把该文件当做Avro数据文件进行解析。Avro从数据文件读取的每个元素被称为datum，每个datum都遵循数据模式定义的数据结构。

本例中，我们没有使用明确的模式，而是把每个datum读入GenericRecord类，并使用字段名从中提取各个字段的值。

Avro中的GenericRecord类非常灵活，它可被用于封装任何数据结构，比如示例中用到的UFO-sighting类型。Avro也支持原生类型，如整型、浮点型、布尔型，以及其他结构类型，如数组和枚举。在这些例子中，我们将记录用作最常用的结构，但这只是为了方便。

在MapReduce中使用Avro

Avro对MapReduce的支持主要体现在，Avro对若干个常见类进行了改写，实现了Avro特有变种。我们通常希望，Hadoop通过改写InputFormat和OutputFormat类实现对新数据文件格式的支持。我们将使用**AvroJob**、**AvroMapper**和**AvroReducer**代替非Avro版本的类。AvroJob希望使用Avro数据文件作为输入和输出，所以我们使用输入和输出的Avro数据模式对AvroJob进行配置，而不再指定输入和输出数据的格式类型。

AvroMapper、AvroReducer与通用mapper、reducer的主要区别在于使用的数据类型。默认情况下，Avro的输入输出是单一的，而Mapper和Reducer类要求使用键值对作为输入输出类型。为此，Avro引入了Pair类，该类通常用于输出处理过程中产生的临时键值数据。

Avro还支持**AvroKey**和**AvroValue**，它们可以封装其他数据类型，但是下面示例中没有用到这两个类型。

5.16　实践环节：在 MapReduce 中统计 UFO 形状

本节，我们将编写一个mapper，它以前面定义好的UFO目击事件记录为输入。它会输出相应的UFO形状和值1，reducer会利用mapper的输出生成一个新的Avro数据类型，它包括了每个UFO形状出现的次数。执行下列步骤完成该任务。

(1) 把sightings.avro文件拷贝到HDFS。

```
$ hadoopfs -mkdiravroin
$ hadoopfs -put sightings.avroavroin/sightings.avro
```

(2) 把下列代码保存为AvroMR.java文件。

```
import java.io.IOException;
```

```
import org.apache.avro.Schema;
import org.apache.avro.generic.*;
import org.apache.avro.Schema.Type;
import org.apache.avro.mapred.*;
import org.apache.avro.reflect.ReflectData;
import org.apache.avro.util.Utf8;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.io.* ;
import org.apache.hadoop.util.*;

//定义输出记录的数据结构
class UFORecord
{
UFORecord()
    {
    }

    public String shape ;
    public long count ;
}

public class AvroMR extends Configured    implements Tool
{
// 创建map输出的模式
    public static final Schema PAIR_SCHEMA =
Pair.getPairSchema(Schema.create(Schema.Type.STRING),
Schema.create(Schema.Type.LONG));
// 创建reduce输出的模式
    public final static Schema OUTPUT_SCHEMA =
ReflectData.get().getSchema(UFORecord.class);

    @Override
    public int run(String[] args) throws Exception
    {
JobConfconf = new JobConf(getConf(), getClass());
conf.setJobName("UFO count");

        String[] otherArgs = new GenericOptionsParser(conf, args).
getRemainingArgs();
        if (otherArgs.length != 2)
        {
System.err.println("Usage: avro UFO counter <in><out>");
System.exit(2);

        }

FileInputFormat.addInputPath(conf, new Path(otherArgs[0]));
        Path outputPath = new Path(otherArgs[1]);
FileOutputFormat.setOutputPath(conf, outputPath);
outputPath.getFileSystem(conf).delete(outputPath);
        Schema input_schema =
```

5

```
Schema.parse(getClass().getResourceAsStream("ufo.avsc"));
AvroJob.setInputSchema(conf, input_schema);
AvroJob.setMapOutputSchema(conf,
Pair.getPairSchema(Schema.create(Schema.Type.STRING),
Schema.create(Schema.Type.LONG)));

AvroJob.setOutputSchema(conf, OUTPUT_SCHEMA);
AvroJob.setMapperClass(conf, AvroRecordMapper.class);
AvroJob.setReducerClass(conf, AvroRecordReducer.class);
conf.setInputFormat(AvroInputFormat.class) ;
JobClient.runJob(conf);

        return 0 ;
    }

    public static class AvroRecordMapper extends
AvroMapper<GenericRecord, Pair<Utf8, Long>>
    {
        @Override
        public void map(GenericRecord in, AvroCollector<Pair<Utf8,
Long>> collector, Reporter reporter) throws IOException
        {
            Pair<Utf8,Long> p = new Pair<Utf8,Long>(PAIR_SCHEMA) ;
Utf8 shape = (Utf8)in.get("shape") ;
            if (shape != null)
            {
p.set(shape, 1L) ;
collector.collect(p);
            }
        }
    }

    public static class AvroRecordReducer extends
AvroReducer<Utf8,
Long, GenericRecord>
    {

        public void reduce(Utf8 key, Iterable<Long> values,
AvroCollector<GenericRecord> collector,
            Reporter reporter) throws IOException
        {

            long sum = 0;
            for (Long val : values)
            {
                sum += val;
            }

GenericRecord value = new
GenericData.Record(OUTPUT_SCHEMA);



value.put("shape", key);
```

```
value.put("count", sum);

collector.collect(value);
        }
    }

    public static void main(String[] args) throws Exception
    {
int res = ToolRunner.run(new Configuration(), new AvroMR(),
args);
System.exit(res);
    }
}
```

(3) 编译并运行作业。

```
$ javacAvroMR.java
$ jar -cvfavroufo.jar *.class ufo.avsc
   $ hadoop jar ~/classes/avroufo.jarAvroMRavroinavroout
```

(4) 检查输出路径。

```
$ hadoopfs -lsavroout
Found 3 items
-rw-r--r--    1 … /user/hadoop/avroout/_SUCCESS
drwxr-xr-x    - hadoopsupergroup    0 … /user/hadoop/
avroout/_logs
-rw-r--r--    1 …    /user/hadoop/avroout/part-00000.avro
```

(5) 把输出文件拷贝到本地文件系统。

```
$ hadoopfs -get /user/hadoop/avroout/part-00000.avroresult.avro
```

原理分析

我们创建了Job类并检查其各个组件。实际上，mapper和reducer类的逻辑很简单：mapper类只是从shape字段提取相应值并输出，同时输出值1；然后，reducer统计每个UFO形状出现的总次数。我们关注的是，为Mapper和Reducer定义的输入输出类型，以及作业的配置方式。

Mapper类的输入类型为GenericRecord，输出类型为Pair。相应地，Reducer类的输入类型为Pair，输出类型为GenericRecord。

传给Mapper类的GenericRecord类对datum（输入文件中的UFO目击事件记录）进行了封装。这就是Mapper类能够通过**Shape**字段名获取对应值的原因。

回忆一下，创建GenericRecords时，可以明确地或隐含地指定数据模式。这两种情况下，都可以将数据模式编码在数据文件中。对于Reducer类用到的GenericRecords输出，我们通过

其他方式为其创建了一个模式。

在上述代码中，我们创建了UFORecord类，并在其运行时使用Avro反射动态生成模式。之后，我们可以使用该模式创建GenericRecord类，该类专门用于封装那种特定的数据类型。

我们使用Avro Pair类型在Mapper和Reducer之间存储键值对。这样，Mapper和Reducer类实现的逻辑与第2章中的WordCount程序的逻辑完全相同：Mapper类输出每个形状的出现次数，Reducer类累加每个形状的出现次数作为最终输出。

除Mapper和Reducer类的输入输出外，还要对处理Avro数据的作业进行一些独特的配置。

```
Schema input_schema = Schema.parse(getClass().
getResourceAsStream("ufo.avsc")) ;
AvroJob.setInputSchema(conf, input_schema);
AvroJob.setMapOutputSchema(conf,           Pair.getPairSchema(Schema.
create(Schema.Type.STRING), Schema.create(Schema.Type.LONG)));

AvroJob.setOutputSchema(conf, OUTPUT_SCHEMA);
AvroJob.setMapperClass(conf, AvroRecordMapper.class);
AvroJob.setReducerClass(conf, AvroRecordReducer.class);
```

这些配置元素说明了模式对Avro的重要性。尽管不定义模式也可完成数据处理任务，我们必须设置输入输出数据的模式类型。Avro会验证输入输出数据是否与指定的模式相吻合，这在某种程度上也保证了数据类型安全。对其他配置元素，比如Mapper和Reducer的配置选项，我们通过AvroJob类而非通用Job类进行设置。设置完成后，MapReduce框架会根据这些设置执行作业。

本例也是我们首次明确实现Tool接口。在运行Hadoop命令行程序时，有一系列参数（比如-D）对多个子命令是通用的。假如作业类像上节提到的那样实现了Tool接口，它会自动访问通过命令行传入的标准参数。这个方法可有效避免不少代码重复。

5.17　实践环节：使用 Ruby 检查输出数据

既然作业已生成了输出数据，我们将使用Ruby检查这些输出数据。

(1) 将下列代码保存为read.rb文件。

```
require 'rubygems'
require 'avro'

file = File.open('res.avro', 'rb')
reader = Avro::IO::DatumReader.new()
dr = Avro::DataFile::Reader.new(file, reader)

dr.each {|record|
print record["shape"]," ",record["count"],"\n"
}
dr.close
```

(2) 检查作业输出的结果文件。

```
$ ruby read.rb
blur 1
cylinder 1
diamond 2
formation 1
light 3
saucer 1
```

原理分析

和以前一样，我们不会分析Ruby的Avro API。示例程序创建了一个打开Avro数据文件的Ruby脚本，循环读取每个datum并基于明确的字段名输出结果。请注意，该脚本没有访问数据文件的模式，数据文件头部提供的信息足以满足Ruby脚本获取各个字段的需要。

5.18 实践环节：使用 Java 检查输出数据

为了证明可使用多种语言访问Avro数据，我们将使用Java语言显示作业输出。

(1) 将下列代码保存为`OutputRead.java`文件。

```
import java.io.File;
import java.io.IOException;
import org.apache.avro.file.DataFileReader;
import org.apache.avro.generic.GenericData;
import org.apache.avro. generic.GenericDatumReader;
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.io.DatumReader;

public class OutputRead
{

    public static void main(String[] args) throws IOException
    {
        String filename = args[0] ;

        File file=new File(filename) ;
DatumReader<GenericRecord> reader= new
GenericDatumReader<GenericRecord>();
DataFileReader<GenericRecord>dataFileReader=new
DataFileReader<GenericRecord>(file,reader);

        while (dataFileReader.hasNext())
        {
GenericRecord result=dataFileReader.next();
            String output = String.format("%s %d",
```

```
result.get("shape"), result.get("count")) ;
System.out.println(output) ;
        }
    }
}
```

(2) 编译并运行程序。

```
$ javacOutputResult.java
$ java OutputResultresult.avro
blur 1
cylinder 1
diamond 2
formation 1
light 3
saucer 1
```

原理分析

我们加入该示例的目的是说明可使用多种语言读取Avro数据。上述代码与之前讲到的InputRead类非常相似，唯一的区别在于，字段名用于展示从数据文件读取的每个datum。

一展身手：Avro对图的支持

前面曾提到过，我们努力简化GraphPath类中图的表示方法。但扁平的文本行与图对象之间存在映射关系，维护它们之间的转换需要一定的开销。

由于Avro支持复杂类型的嵌套，它天生就支持以更接近运行时对象的方式表示节点。修改GraphPath类作业，从表示节点的datum组成的Avro数据文件中读写图。读者可以使用下面的示例模式，但还可以对其进行改进。

```
{ "type": "record",
  "name": "Graph_representation",
  "fields" : [
{"name": "node_id", "type": "int"},
  {"name": "neighbors", "type": "array", "items:"int" },
  {"name": "distance", "type": "int"},
 {"name": "status", "type": "enum",
"symbols": ["PENDING", "CURRENT", "DONE"
},]
]
}
```

继续研究Avro

本章的案例研究仅提到了Avro的部分特性。我们重点介绍了Avro作为静态数据表示方法的价

值。它还可以用在RPC（remote procedure call，远程过程调用）框架中，Hadoop 2.0可以选择它作为默认的RPC格式。我们没有使用Avro的代码生成工具，它可以产生特定领域的API。我们也没有讨论Avro对模式演变的支持，比如，它可以在新记录中加入新字段而不会影响老datum或破坏原有客户。这种技术在未来应该会得到更广泛的应用。

5.19 小结

本章通过3个案例研究，重点介绍了Hadoop的一些高级特性及其广泛的生态系统。我们特别讨论了不同类型联结的本质问题以及它们在Hadoop的哪个阶段出现,如何相对简单却又高效地实现reduce端联结，以及如何把数据送进Distributed Cache避免map端的完整联结。

接着，我们学习了如何实现完整的map端联结，但它需要对输入数据进行大量处理。如果经常需要进行联结操作，应当研究其他工具，比如Hive和Pig。还介绍了如何考虑像图这样的复杂类型以及在MapReduce中用怎样的方法表示图。

我们还学习了将图算法分成多阶段MapReduce作业的技术，语言无关的数据类型的重要性，如何使用多种语言操作Avro数据，以及如何将Avro扩展到MapReduce API（它允许使用结构类型作为MapReduce作业的输入输出）。

现在,我们对Hadoop MapReduce编程的介绍就结束了。第6章和第7章将讨论如何管理Hadoop集群以及如何扩展其规模。

第 6 章

故障处理

6

Hadoop的一个重要特性是故障恢复能力，本章将重点学习Hadoop的容错机制。

本章包括以下内容：

- Hadoop如何处理DataNode和TaskTracker的故障；
- Hadoop如何处理NameNode和JobTracker的故障；
- 硬件故障对Hadoop的影响；
- 如何处理由软件缺陷引发的任务故障；
- 错误数据如何引发任务故障以及应对方法。

同时，我们将更深入地理解Hadoop的各个部件是如何协同工作的，并通过实践发现一些好的做法。

6.1 故障

在许多技术领域中，很少见到技术文档会用大量篇幅来介绍故障处理方法。通常，人们认为只有技术专家才会对故障处理技术感兴趣。在Hadoop中，故障处理的位置更靠前而且是一个核心问题。Hadoop架构及设计的前提，就是作业执行过程中会经常发生故障，并且故障在所难免。

6.1.1 拥抱故障

近年来，与传统的害怕发生故障的心态不同，出现了一种被称为“拥抱故障”的全新理念。之前人们寄希望于不发生故障，现在则是接受故障是无法避免的事实，并知道故障发生时系统和流程应如何应对。

6.1.2 至少不怕出现故障

这是一种理念上的延伸，因此，本章目的在于让读者明白如何应对Hadoop系统中的各类故障，不至于在发生故障时手足无措。我们将通过杀死正在运行的群集中的进程，故意造成软件失败，向作业推送错误数据等多种方式尽可能制造破坏。

6.1.3 严禁模仿

通常，由于测试实例被滥用，业务系统已对其进行了防范，因此并不会对业务系统造成破坏。但是，我们仍不主张对运行中的Hadoop集群实施本章给出的破坏实例，尽管除了一两个非常特殊的案例，其他的都没有危险。我们的目标是了解各类故障的影响，以便关键业务系统发生故障时，清楚地知道这是不是一个问题。幸运的是，Hadoop可以为我们处理大部分故障。

6.1.4 故障类型

我们通常将故障划分为以下5类。

- 节点故障，也就是DataNode或TaskTracker进程的故障。
- 集群主节点的故障，也就是NameNode或JobTracker进程的故障。
- 硬件故障，包括主机崩溃、硬盘故障等。
- MapReduce作业中由软件错误引发的个别任务故障。
- MapReduce作业中由数据问题引发的个别任务故障。

我们将会在下面各节中依次讲解各种情况。

6.1.5 Hadoop 节点故障

我们将探讨的第一类故障是，个别DataNode或TaskTracker进程意外终止引发的故障。Hadoop声称通过解决硬件故障保证系统可用性，我们认为有理由相信这个说法。确实，随着集群规模增长到成百上千台主机，个别节点发生故障的可能性相当普遍。

在杀死进程之前，我们将介绍一款新工具，并正确设置群集。

1. `dfsadmin`命令

`dfsadmin`命令行工具可以代替HDFS网页用户接口，查看集群的工作状态。其用法如下：

```
$ Hadoop dfsadmin
```

上述命令会列出`dfsadmin`命令的多个选项。为了查看集群状态，我们使用`-report`选项。

该命令给出了集群整体状态的概述，包括额定容量、节点数、文件数和每个节点的具体配置细节。

2. 集群设置、测试文件和数据块大小

后续实践需要一个全分布式的集群，其安装步骤参见本书前述内容。该集群中使用1台主机作为JobTracker和NameNode，其他4台主机用于运行DataNode和TaskTracker进程。



请注意，无需为每个节点配套物理硬件，我们使用虚拟机组建Hadoop集群。

正常情况下，Hadoop集群的数据块大小通常设为64 MB。但该配置并不适合用于测试，因为要将文件切分成多份分布在多点集群上，需要特别大的文件才行。

这就需要我们在配置时减小数据块大小。既然这样，我们将数据块大小配置为4 MB。请对Hadoop `conf`目录下的`hdfs-site.xml`进行如下修改。

```
<property>
<name>dfs.block.size</name>
<value>4194304</value>
;</property>
<property>
<name>dfs.namenode.logging.level</name>
<value>all</value>
</property>
```

第一个属性修改的是数据块大小，第二个属性提升了NameNode的日志级别，因此一些对数据块的操作也会出现在日志文件中。



对本次测试而言，所有这些设置都恰到好处；但它们很少用在产品集群的配置中。尽管在调查非常难的问题时，可能需要调高NameNode的日志级别，但是绝对不可能将数据块设为4 MB这么小。虽然Hadoop可处理较小的数据块，但是小数据块会影响Hadoop的运行效率。

我们还需要一个大小适中的测试文件，它需要被切分为多个4 MB大小的数据块。实际上，我们不会用到该文件的内容，所以文件类型在这里并不重要。但是，为了给后几节提供便利，你应该把手头上最大的文件复制到HDFS。此处，我们使用了一个CD ISO映像文件。

```
$ Hadoop fs -put cd.iso file1.data
```

3. Elastic MapReduce的容错机制

为了更明确地介绍故障细节，本书使用本地Hadoop集群作为例子。EMR的容错机制与本地

集群完全相同，因此这里讲到的故障情景既适用于本地Hadoop集群也适用于EMR托管的集群。

6.2　实践环节：杀死 DataNode 进程

首先，杀掉一个DataNode进程。回忆一下，HDFS集群的每台主机都运行着一个DataNode进程，它负责管理HDFS文件系统的数据块。默认情况下，Hadoop中数据块的复制因子为3，因此，我们希望单个DataNode的故障不会直接影响到Hadoop的可用性。当然，单个DataNode的故障会导致某些数据块的副本数量暂时小于复制因子门限值。执行下列步骤杀死DataNode进程。

(1) 首先，查看集群的原始状态并检查所有部件是否正常工作。使用`dfsadmin`命令实现这个目的。

```
$ Hadoop dfsadmin -report
Configured Capacity: 81376493568 (75.79 GB)
Present Capacity: 61117323920 (56.92 GB)
DFS Remaining: 59576766464 (55.49 GB)
DFS Used: 1540557456 (1.43 GB)
DFS Used%: 2.52%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0

-------------------------------------------------
Datanodes available: 4 (4 total, 0 dead)


Name: 10.0.0.102:50010
Decommission Status : Normal
Configured Capacity: 20344123392 (18.95 GB)
DFS Used: 403606906 (384.91 MB)
Non DFS Used: 5063119494 (4.72 GB)
DFS Remaining: 14877396992(13.86 GB)
DFS Used%: 1.98%
DFS Remaining%: 73.13%
Last contact: Sun Dec 04 15:16:27 PST 2011
...
```

现在登录到其中一个节点，使用`jps`命令查看DataNode进程的进程ID：

```
$ jps
2085 TaskTracker
2109 Jps
1928 DataNode
```

(2) 使用DataNode的**process ID**（PID）杀死进程。

```
$ kill -9    1928
```

(3) 检查DataNode进程是否仍在运行。

```
$ jps
```

6

```
2085 TaskTracker
```

(4) 再次使用dfsadmin命令查看集群状态。

```
$ Hadoop dfsadmin -report
Configured Capacity: 81376493568 (75.79 GB)
Present Capacity: 61117323920 (56.92 GB)
DFS Remaining: 59576766464 (55.49 GB)
DFS Used: 1540557456 (1.43 GB)
DFS Used%: 2.52%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0

-------------------------------------------------
Datanodes available: 4 (4 total, 0 dead)
...
```

(5) 要重点关注包含每个节点的数据块数量、活跃节点数和最后通信时间的行。死亡节点的最后通信时间距现在差不多10分钟的时候，经常使用$ Hadoop dfsadmin -report查看集群状态，直到数据块的数量和活跃节点数发生变化。

```
$ Hadoop dfsadmin -report
Configured Capacity: 61032370176 (56.84 GB)
Present Capacity: 46030327050 (42.87 GB)
DFS Remaining: 44520288256 (41.46 GB)
DFS Used: 1510038794 (1.41 GB)
DFS Used%: 3.28%
Under replicated blocks: 12
Blocks with corrupt replicas: 0
Missing blocks: 0
-------------------------------------------------
Datanodes available: 3 (4 total, 1 dead)
...
```

(6) 重复上述过程，直到再次输出“Under replicated blocks：0”。

```
$ Hadoop dfsadmin -report
...
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0
-------------------------------------------------
Datanodes available: 3 (4 total, 1 dead)
...
```

原理分析

从较高的视角来看，似乎并无难解之处。Hadoop确认集群中少了一个节点并开展工作解决该问题。但是，为了解决该问题，Hadoop做了大量工作。

当我们杀死Datanode进程时，该主机上的进程已经不再提供服务或接收数据块进行读/写操作。但是，当时我们并没有访问文件系统，那么NameNode进程是怎么知道某个特定的DataNode已经无法提供服务了呢？

NameNode和DataNode之间的通信

答案是NameNode和Datanode进程之间保持着频繁通信。虽然我们曾经提到过一两次它们之间的通信，却从来没有详细地解释。这个过程由DataNode发出的一系列固定心跳消息完成，这些消息向NameNode报告其当前状态及保存的数据块。作为回应，NameNode向DataNode发出指令，如通知DataNode新建一个文件或命令它从另一节点获取数据块。

当NameNode进程启动起来并开始接收DataNode的状态信息时，整个通信过程就已经开始。回想一下，每个DataNode知道其NameNode的位置，并不断向其发送状态报告。这些消息列出了各个DataNode保存的数据块。基于这些信息，NameNode能够在数据块和文件、路径之间建立完整映射，这样NameNode就会知道文件由哪些数据块组成以及这些数据块存放在哪些节点之上。

NameNode进程对每个DataNode最后一次发送心跳信息的时间进行监测，一旦该时间超过门限值之后，NameNode就会认定某个DataNode无法继续使用，并将其标记为dead。



DataNode被认定为dead的准确门限值并不是HDFS的一个可配置属性。相反，它是通过其他几个属性计算得出的，比如心跳间隔属性在该值的计算中起决定作用。稍后我们会看到，MapReduce的对应属性值的确定稍微简单一些，TaskTracker的超时时间由一个配置属性所控制。

6

一旦某个DataNode被标记为死亡节点，NameNode进程会确定哪些数据块存储在该节点上，这些数据块的副本数已跌破其复制因子。在默认情况下，被杀死节点上存储的数据块是3个副本中的一个，所以该节点存储的数据块在集群中只剩有2个副本。

在前面例子中，我们发现有12个数据块的副本数量小于其复制因子，也就是说，这些数据块在整个集群中的副本数量无法达到其目标数量。当NameNode进程明确了副本数过低的数据块后，它分配其他DataNode从现有副本驻留主机复制这些数据块。在这种情况下，需要重新复制的数据块的数量很少。在活跃集群中，一个节点的故障会导致一段时间的高网络流量，因为NameNode会安排其他节点重新复制死亡节点存储的数据块。

需要注意的是，如果出现故障的节点重新恢复正常运行，该节点存储的数据块在集群中的副本数量可能超过所需数量。在这种情况下，NameNode进程会发出指令，要求删除多余副本。系统随机选择要删除的副本，因此，可能会出现恢复正常运行的节点保留了部分数据块，却删掉了其他数据块的情况。

一展身手：深入研究NameNode的日志

我们已配置NameNode进程记录其所有活动。浏览这些非常详细的日志，并试着从中找出NameNode向DataNode发出的复制要求。

最终输出显示了将副本数量不足的数据块复制到活跃节点后的集群状态。集群的活跃节点数下降到3个，但是所有的数据块都达到了其复制因子的要求。



使用start-all.sh脚本可快速重启所有主机中的死亡节点。该脚本在启动所有部件之前，会智能检测正在运行的服务，这就意味着，它会重启死亡节点却不会对活跃节点产生影响。

6.3 实践环节：复制因子的作用

本节将重复杀死进程的步骤，但这次，要在由4个节点组成的集群中杀死2个DataNode。我们仅会简略描述这个过程，因为它与上一个实践环节的步骤非常相似。

(1) 重启死亡节点并监测集群状态，直到所有节点都被标记为活跃节点。

(2) 选择其中2个DataNode，使用其进程ID杀掉相应进程。

(3) 和上一个实践环节的操作类似，等候大约10分钟之后，通过dfsadmin命令查看集群状态，特别要关注副本数量低于复制因子的数据块的数量。

(4) 等到集群状态稳定之后，输出以下内容。

```
Configured Capacity: 61032370176 (56.84 GB)
Present Capacity: 45842373555 (42.69 GB)
DFS Remaining: 44294680576 (41.25 GB)
DFS Used: 1547692979 (1.44 GB)
DFS Used%: 3.38%
Under replicated blocks: 125
Blocks with corrupt replicas: 0
Missing blocks: 0

-------------------------------------------------
Datanodes available: 2 (4 total, 2 dead)
...
```

原理分析

这个过程与上一个“实践环节”的过程相同。区别在于，由两个DataNode发生故障引起的副

本数量小于复制因子的数据块明显增多，多个数据块的副本数量降为1。因此，读者会看到，由于多个节点故障造成“Under replicated blocks”的值明显增大，之后随着重新复制过程的进行，“Under replicated blocks”的值逐步下降。这些活动也可以从NameNode节点的日志看出。

需要注意的是，虽然Hadoop可以通过重新复制策略将只剩一个副本的数据块复制为两个副本，但这些数据块的数量仍处于under-replicated状态。由于集群中只有两个活跃节点，任何数据块的副本数量都无法达到默认的三个副本的目标。

为了节省版面，我们截断了`dfsadmin`命令的输出。尤其是，一直以来我们都忽略了每个节点的状态信息。然而，让我们看看经过上述操作后的集群中第一个节点的状态。在未杀掉任何DataNode之前，其状态输出如下所示。

```
Name: 10.0.0.101:50010
Decommission Status : Normal
Configured Capacity: 20344123392 (18.95 GB)
DFS Used: 399379827 (380.88 MB)
Non DFS Used: 5064258189 (4.72 GB)
DFS Remaining: 14880485376(13.86 GB)
DFS Used%: 1.96%
DFS Remaining%: 73.14%
Last contact: Sun Dec 04 15:16:27 PST 2011
```

在杀掉一个DataNode节点，所有数据块都按需重新复制之后，其状态输出如下所示。

```
Name: 10.0.0.101:50010
Decommission Status : Normal
Configured Capacity: 20344123392 (18.95 GB)
DFS Used: 515236022 (491.37 MB)
Non DFS Used: 5016289098 (4.67 GB)
DFS Remaining: 14812598272(13.8 GB)
DFS Used%: 2.53%
DFS Remaining%: 72.81%
Last contact: Sun Dec 04 15:31:22 PST 2011
```

需要注意的是，该节点上的本地DFS存储量增加了。这并不足为奇。由于集群出现了一个死亡节点，集群中的其他节点需要增加一些额外的数据块副本，这就造成了每个节点上已用存储空间的增加。

在集群中的另外两个DataNode被杀死之后，第一个节点的状态如下所示。

```
Name: 10.0.0.101:50010
Decommission Status : Normal
Configured Capacity: 20344123392 (18.95 GB)
DFS Used: 514289664 (490.46 MB)
Non DFS Used: 5063868416 (4.72 GB)
DFS Remaining: 14765965312(13.75 GB)
DFS Used%: 2.53%
DFS Remaining%: 72.58%
Last contact: Sun Dec 04 15:43:47 PST 2011
```

集群中的死亡节点增加到了2个，似乎剩余的活跃节点应该消耗更多的本地存储空间，但本例中的情况并非如此，其原因仍与复制因子有着密切关系。

如果集群由4个节点组成，其复制因子为3，那么其中3个活跃节点会分别存储每个数据块的3个副本。假如死掉1个节点，存储于其余3个节点的数据块不受影响，而存储在该节点的数据块需要新建一个副本。但是，由于只有3个活跃节点，每个节点需要为每个数据块保存1个副本。假如第2个节点发生故障，会导致所有数据块的副本数量都小于复制因子，同时Hadoop找不到地方存放额外副本。因为剩余的存活节点已经为每个数据块保存了1个副本，它们的空间使用率不会继续增加。

6.4　实践环节：故意造成数据块丢失

很明显，下一步我们将快速杀掉3个DataNode。



之前我们曾提到，有些操作不能在产品集群上进行，这个例子就是这种情况。尽管这些步骤如果操作得当，不会引起数据丢失，但却会导致现有数据在某段时间内无法使用。

使用以下步骤连续杀死3个DataNode。

(1) 使用下列命令重启所有节点。

```
$ start-all.sh
```

(2) 等待，直到Hadoop的 `dfsadmin -report`命令显示有4个活跃节点。

(3) 把测试文件的新副本`file1.new`放到HDFS上。

```
$ Hadoop fs -put file1.data file1.new
```

(4) 登录到集群中的3台主机并杀死每台主机上的DataNode进程。

(5) 等候10分钟，之后通过`dfsadmin`命令监视集群状态，直到集群状态如下所示。

```
...
Under replicated blocks: 123
Blocks with corrupt replicas: 0
Missing blocks: 33


-------------------------------------------------
 Datanodes available: 1 (4 total, 3 dead)
...
```

(6) 尝试从HDFS获取测试文件`file1.new`。

```
$ hadoop fs -get file1.new    file1.new
11/12/04 16:18:05 INFO hdfs.DFSClient: No node available for
```

```
block: blk_1691554429626293399_1003 file=/user/hadoop/file1.new
11/12/04 16:18:05 INFO hdfs.DFSClient: Could not obtain block
blk_1691554429626293399_1003 from any node:    java.io.IOException:
No live nodes contain current block
...
get: Could not obtain block: blk_1691554429626293399_1003 file=/
user/hadoop/file1.new
```

(7) 使用start-all.sh脚本重启所有死亡节点。

```
$ start-all.sh
```

(8) 不断监视数据块状态。

```
$ Hadoop dfsadmin -report | grep -i blocks
Under replicated blockss: 69
Blocks with corrupt replicas: 0
Missing blocks: 35
$ Hadoop dfsadmin -report | grep -i blocks
Under replicated blockss: 0
Blocks with corrupt replicas: 0
Missing blocks: 30
```

(9) 等到输出Missing blocks: 0，然后将测试文件file1.new拷贝到本地文件系统。

```
$ Hadoop fs -get file1.new file1.new
```

(10) 对获取的文件和原始文件进行MD5校验。

```
$ md5sum file1.*
f1f30b26b40f8302150bc2a494c1961d    file1.data
f1f30b26b40f8302150bc2a494c1961d    file1.new
```

原理分析

在重启被杀死进程之后，我们将测试文件拷贝到HDFS。严格意义上来讲，将测试文件拷贝到HDFS并不是必需的，我们也可以使用HDFS上的已有文件，但由于数据块分别存储在不同的主机，使用原始副本得出的结果更具代表性。

之后，我们按照前述步骤杀死3个DataNode，并等候HDFS的响应。与前面几个例子不一样的是，杀死这么多节点意味着某些数据块的所有副本都存储在被杀死的节点上。如我们所见，结果正是如此：仅剩1个节点的集群中有100多个数据块的副本数量低于复制因子（很明显，这些数据块仅剩1个副本），同时丢了33个数据块。

讨论数据块有点抽象，所以我们尝试获取测试文件。因为我们知道，由于缺少33个数据块造成测试文件不完整，就像是有33个洞。访问测试文件以失败告终，因为Hadoop无法找到缺失的数据块，而这些数据块在文件传送过程中是必需的。

接下来我们重新启动所有节点并再次尝试获取测试。这一次终于成功了，但我们采取了一个额外的验证措施，对获得的文件进行MD5校验，以确认它与原始文件完全相同——结果确实如此。

从本例可以看出，虽然节点故障可能会导致数据无法访问，但节点恢复之后，这个暂时性的数据缺失问题就不复存在。这点特别重要。

1. 数据丢失的可能性

不要从本例中轻易得出Hadoop集群不会丢失数据这样的结论。一般来讲，很难出现数据丢失这样的情况，但灾难往往喜欢以错误的方式突然出现。

如上例所示，不少于复制因子的多个节点同时发生故障有可能导致数据块缺失。在由4台主机组成的示例集群中，3个死亡节点导致数据块缺失的可能较高。在由1000台主机组成的集群中，这个可能性相对较低但仍然存在。随着集群规模增大，故障率也随之增大，狭小的时间窗口内，3个节点同时发生故障的可能性越来越低。因此，它带来的影响也减小了，但多个节点接连发生故障总会带来数据丢失的风险。

另一个隐蔽的问题是反复故障或局部故障。例如，整个集群的不稳定电源会导致节点反复关机和重启。 Hadoop为了达成复制目标，可能会不断要求恢复运行的主机复制那些副本数量少于复制因子的数据块，电源问题导致的突然关机又会造成这些任务的中途失败。这样一系列事件也会提高潜在的数据丢失风险。

最后，不要忘了人为因素。即使将复制因子设置为集群中的主机数量，它保证了每个节点上都存储了一份数据副本，但在用户意外删除文件或目录时也无济于事。

结论是，由系统故障造成的数据丢失的可能性很小，但由无法避免的人工操作造成的数据丢失的可能性仍然存在。复制数据并不是一个完整的备份方案。用户必须充分认识到待处理数据的重要性以及本节讨论的数据丢失对我们造成的影响。



实际上，Hadoop集群中最严重的数据丢失是由NameNode和文件系统损坏造成的。我们将在下一章比较详细地讨论这一话题。

2. 数据块损坏

DataNode发出的状态报告中也包括了已损坏数据块的数量，这一点我们之前从未提及。DataNode将数据块初次写入HDFS时，同时将一个包含本数据块密码校验和的隐藏文件写入相同目录。默认情况下，DataNode为每512字节生成一个校验和。

每当客户端读取数据块时，同时也会读取校验和列表。客户端会计算已读取数据的校验和，并将其与获取的校验和列表进行对比。如果两个校验和不一致，该DataNode节点上的数据块被标

记为损坏数据，客户端会获取另一个副本。得知数据块已损坏之后，NameNode会调度该DataNode基于现有的未损坏副本生成一个新的副本。

如果你认为不会发生数据损坏的情况，想一下出现故障的内存、硬盘、存储控制器，或单台主机的其他问题，它们都可能在初次写入数据块或读取数据块时导致数据块损坏。这些情况非常罕见。存储同一数据块副本的所有DataNode发生相同数据损坏的可能性微乎其微。但是，请记住，正如前面提到的，复制并不是一个完整的备份方案，如果用户需要确保数据100%可用，可能需要考虑在集群之外备份数据。

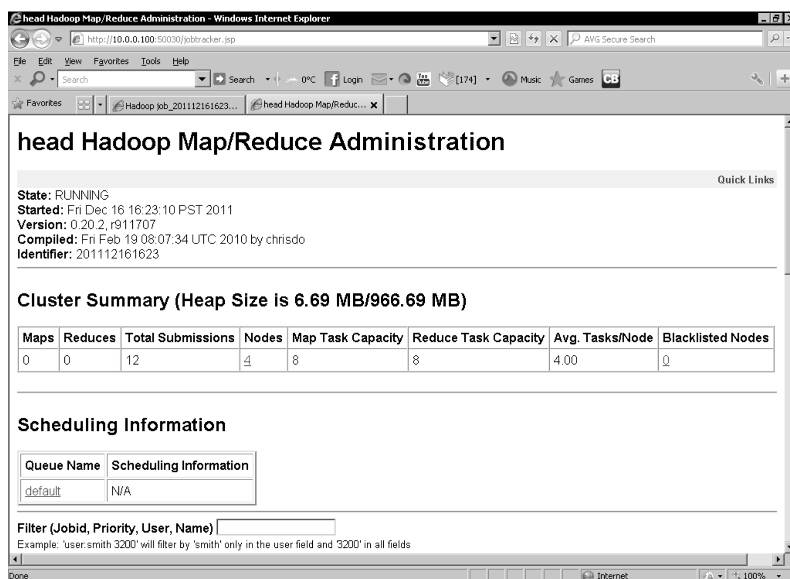
6.5 实践环节：杀死 TaskTracker 进程

我们已经对HDFS及其DataNode故障进行了太多的讨论，现在来看看杀死某些TaskTracker进程会对MapReduce造成什么损坏。

虽然MapReduce中有一个`mradmin`命令与HDFS的`dfsadmin`命令类似，但它无法提供类似HDFS的状态报告。因此，我们使用MapReduce的Web用户接口（默认情况下，该接口位于JobTracker主机的500 70端口）监测MapReduce集群的状态。

执行下列步骤。

(1) 通过`start-all.sh`脚本启动所有部件，之后将浏览器指向MapReduce网页用户接口。页面与下图相似。



(2) 启动一个长期运行的MapReduce作业，带有较大参数的计算圆周率的示例程序就很合适。

```
$ Hadoop jar Hadoop/Hadoop-examples-1.0.4.jar pi 2500 2500
```

(3) 登录到集群中的一个节点，使用jps命令确定TaskTracker的进程ID。

```
$ jps
21822 TaskTracker
3918 Jps
3891 DataNode
```

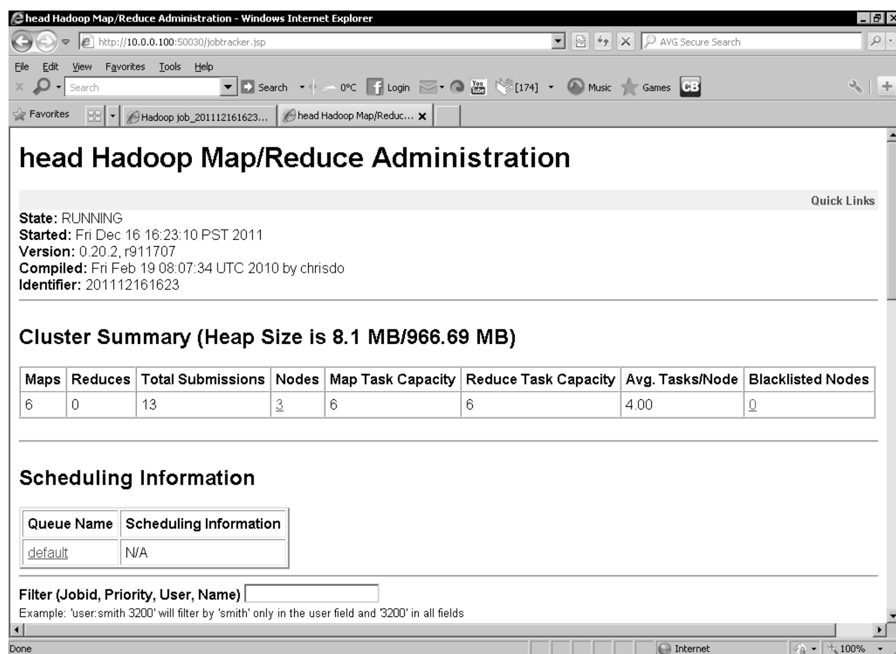
(4) 使用下列命令杀死TaskTracker进程。

```
$ kill -9 21822
```

(5) 确认TaskTracker不再处于运行状态。

```
$jps
3918 Jps
3891 DataNode
```

(6) 返回MapReduce网页用户接口，10分钟后，你应该看到节点数、可用的map/reduce slot数量发生了变化，如下图所示。

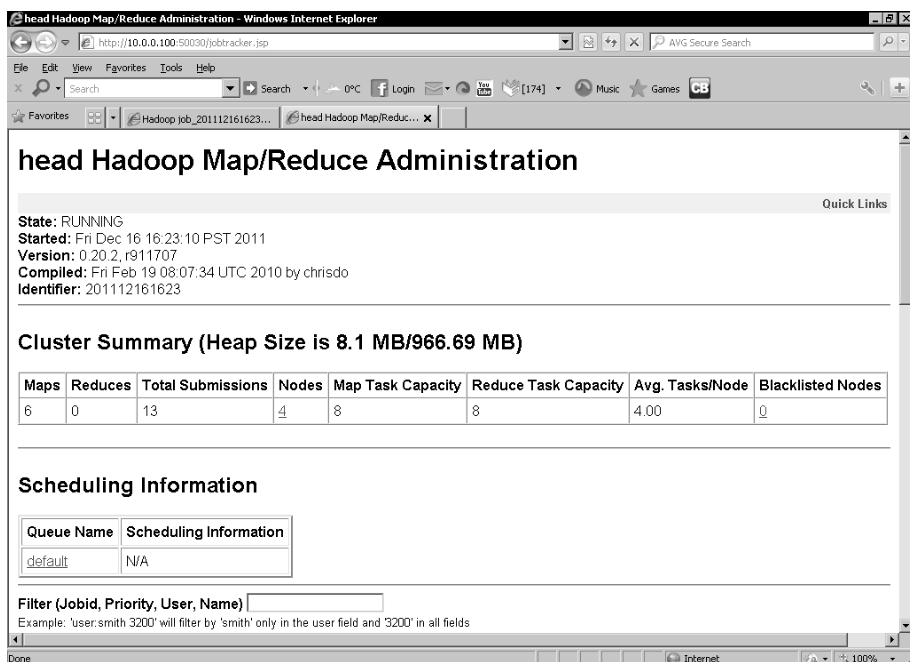


(7) 在原来的窗口中查看作业进度，作业应该处于proceeding状态，虽然其运行缓慢。

(8) 重启被杀死的TaskTracker进程。

```
$ start-all.sh
```

(9) 查看MapReduce网页用户接口。片刻之后，节点数又恢复到了原来的数量，如下图所示。



原理分析

MapReduce网页接口提供了许多关于集群和作业的信息。本节我们感兴趣的重要的数据是Cluster Summary（集群摘要信息），包括Maps、Ruduces（目前正在执行的map和reduce任务数）、Total Submissions（已提交的作业总数）、Nodes（节点数）、Map/Reduce Task Capacity（map和reduce任务容量），还有Blacklisted Nodes（列入黑名单的节点数）。

JobTracker进程和TaskTracker进程之间的关系与NameNode和DataNode之间的关系有较大区别，但它们都使用了类似的心跳机制。

TaskTracker进程频繁向JobTracker发送心跳信息，但与DataNode向NameNode报告数据块的状态信息不同，TaskTracker的心跳信息包含任务进度及可用空间。每个节点都有一个可配置的map和reduce slot（默认值为2），这就解释了为什么我们在第一个网页用户接口页面看到集群由4个节点组成，却有8个map和reduce slot。

当我们杀死TaskTracker进程时，JobTracker进程收不到其心跳信息。在用户设置的时间过后，节点被认定为死亡节点，集群容量相应地减少，这个变化反映在网页用户接口。



用户可通过修改mapred-site.xml文件的mapred.tasktracker.expiry.interval属性配置TaskTracker进程的超时时间值，该值被用于认定节点是否死亡。

当一个TaskTracker进程被标记为死亡节点时，JobTracker进程会认为正在执行的任务失败，并将它们重新分配给集群中的其他节点。尽管某个节点被杀死了，但整个作业却最终执行成功，这也从侧面验证了上述内容。

重新启动TaskTracker进程后，它会给JobTracker发送心跳消息，JobTracker会把它标记为存活节点并再次将其纳入MapReduce集群。我们在最后那张截图中看到，集群节点数和任务slot容量重新恢复到了其原始值，这表明重启后的TaskTracker重新成了MapReduce集群的一部分。

1. DataNode故障和TaskTracker故障对比

我们不会执行类似的杀死2个或3个TaskTracker节点的操作，因为任务执行运行机制表明，个别的TaskTracker故障相对来讲不是很重要。因为TaskTracker进程由JobTracker控制和协调，个别TaskTracker发生故障只会减少集群可并发运行的任务数，除此之外不会产生其他直接影响。如果一个TaskTracker实例失败，JobTracker会调度集群中一个运转正常的TaskTracker进程重新运行失败的任务。JobTracker可以随意在集群中重新安排任务，因为TaskTracker是无状态的，单个TaskTracker故障不会影响该作业的其他部分。

相反，DataNode本质上是有状态的，某个DataNode故障可以影响HDFS上存储的持久数据，可能导致无法访问这些数据。

上述内容强调了各类节点的性质及其在整体Hadoop框架中的相互关系。DataNode管理数据，TaskTracker对DataNode存储的数据进行读写。即使每个TaskTracker都发生灾难性故障，都不会影响到HDFS的功能。而NameNode进程中的类似故障会导致活跃的MapReduce集群无法使用（除非MapReduce的配置允许其使用其他存储系统）。

2. 永久故障

截至目前，我们假设死亡节点可在相同的物理主机重新启动。但是假如物理主机发生了致命故障导致死亡节点无法重启，那该怎么办呢？答案很简单，将该主机从slaves文件删掉，Hadoop就不再启动那台主机上的DataNode或TaskTracker。假如你有一台替代主机，将新主机加入到相同文件并运行`start-all.sh`脚本。



请注意，只能通过`start/stop`和`slaves.sh`脚本访问slaves文件。用户只需在运行这些命令的主机上更新该文件，而无需在每个节点都进行同样操作。实际上，运行这些命令的主机通常是个专用的主节点，或者NameNode或JobTracker进程所在的主机。我们将在第7章介绍这些设置。

杀死集群主节点

虽然DataNode进程和TaskTracker进程发生故障产生的影响有大有小，但是相对来讲，个别节点没那么重要。无论哪个TaskTracker或DataNode发生故障都不会引起关注，只有多个节点发生故障才会给系统带来问题，尤其是多个节点接二连三地发生故障。但是，我们只有一个JobTracker和NameNode，让我们看看它们发生故障会造成什么后果。

6.6　实践环节：杀死 JobTracker

首先，我们将杀死JobTracker进程，我们认为这会影响执行MapReduce作业，但不会对HDFS文件系统造成影响。

(1) 登录到JobTracker主机并杀死该进程。

(2) 启动一个MapReduce示例作业，如计算圆周率的作业或者字数统计作业。

```
$ Hadoop jar wc.jar WordCount3 test.txt output
Starting Job
11/12/11 16:03:29 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9001. Already tried 0 time(s).
11/12/11 16:03:30 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9001. Already tried 1 time(s).
...
11/12/11 16:03:38 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9001. Already tried 9 time(s).

java.net.ConnectException: Call to /10.0.0.100:9001 failed on
connection exception: java.net.ConnectException: Connection
refused
  at org.apache.hadoop.ipc.Client.wrapException(Client.java:767)
  at org.apache.hadoop.ipc.Client.call(Client.java:743)
  at org.apache.hadoop.ipc.RPC$Invoker.invoke(RPC.java:220)
...
```

(3) 执行一些HDFS操作。

```
$ hadoop fs -ls /
Found 2 items
drwxr-xr-x    - hadoop supergroup    0 2011-12-11 19:19 /user
drwxr-xr-x    - hadoop supergroup    0 2011-12-04 20:38 /var
$ hadoop fs -cat test.txt
This is a test file
```

原理分析

在杀死JobTracker进程之后，我们试图启动一个MapReduce作业。在第2章中，我们了解到，

运行作业的主机上的客户端要通过与JobTracker的通信完成作业调度的初始化工作。但本例中，JobTracker已经停止运行，通信过程无法完成，作业以失败告终。

随后，我们执行了一些HDFS命令，再次强调了上节提到的要点：运行异常的MapReduce集群不会对HDFS造成直接影响，所有的客户端都可访问HDFS并执行HDFS命令。

启动替代JobTracker

MapReduce集群的恢复也较为简单。只要重启JobTracker，所有后续MapReduce作业都会成功运行。

请注意，当JobTracker被杀死时，MapReduce框架没有保存任何正在执行的作业的状态信息，所有的未完成作业都需重启。要留意一下HDFS上的临时文件和临时目录，许多MapReduce作业会在HDFS写入一些临时数据，并会在作业运行结束时删掉这些数据。失败的作业（尤其是由JobTracker故障引起的失败作业）可能会留下一些这种文件，用户需要手动清理这些临时数据。

一展身手：在新主机上运行JobTracker

但是，假如JobTracker进程所在的主机发生了致命的硬件错误，无法在该主机上重启JobTracker进程，这时候该怎么办呢？在这种情况下，用户需要在另外一台主机上启动一个新JobTracker进程。这需要对所有节点上的`mapred-site.xml`文件进行修改，更新该文件中的主节点地址，并重启集群。试着完成这些操作。我们将在第7章详细探讨这个话题。

6.7　实践环节：杀死 NameNode 进程

接下来，我们将杀死NameNode进程。我们认为，这将直接导致无法访问HDFS，进而会影响到MapReduce框架，使MapReduce作业无法运行。



不要在正在运行的重要集群中尝试此操作。尽管它带来的影响是短暂的，但整个集群会在一段时间内处于瘫痪状态。

(1) 登录到NameNode主机并列出所有的正在运行的进程。

```
$ jps
2372 SecondaryNameNode
2118 NameNode
2434 JobTracker
5153 Jps
```

(2) 杀死NameNode进程。别理会SecondaryNameNode进程，可以让它继续保持运行。

(3) 尝试访问HDFS文件系统。

```
$ hadoop fs -ls /
11/12/13 16:00:05 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9000. Already tried 0 time(s).
11/12/13 16:00:06 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9000. Already tried 1 time(s).
11/12/13 16:00:07 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9000. Already tried 2 time(s).
11/12/13 16:00:08 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9000. Already tried 3 time(s).
11/12/13 16:00:09 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9000. Already tried 4
time(s).
...
Bad connection to FS. command aborted.
```

(4) 提交MapReduce作业。

```
$ hadoop jar hadoop/hadoop-examples-1.0.4.jar    pi 10 100
Number of Maps    = 10
Samples per Map = 100
11/12/13 16:00:35 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9000. Already tried 0 time(s).
11/12/13 16:00:36 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9000. Already tried 1 time(s).
11/12/13 16:00:37 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9000. Already tried 2 time(s).
...
java.lang.RuntimeException: java.net.ConnectException: Call
to /10.0.0.100:9000 failed on connection exception: java.net.
ConnectException: Connection refused
  at org.apache.hadoop.mapred.JobConf.getWorkingDirectory(JobConf.
java:371)
  at org.apache.hadoop.mapred.FileInputFormat.
setInputPaths(FileInputFormat.java:309)
...
Caused by: java.net.ConnectException: Call to /10.0.0.100:9000
failed on connection exception: java.net.ConnectException:
Connection refused
...
```

(5) 查看正在运行的进程。

```
$ jps
2372 SecondaryNameNode
5253 Jps
2434 JobTracker
Restart NameNode
$ start-all.sh
```

(6) 访问HDFS。

```
$ Hadoop fs -ls /
```

```
Found 2 items
drwxr-xr-x   - hadoop supergroup    0 2011-12-16 16:18 /user
drwxr-xr-x   - hadoop supergroup    0 2011-12-16 16:23 /var
```

原理分析

我们杀死了NameNode进程之后，尝试访问HDFS文件系统。当然，该操作以失败而告终。NameNode宕机之后，集群中就没有接收文件系统命令的服务器了。

随后，我们试着提交一个MapReduce作业，该操作也没有成功。从简略的异常堆栈轨迹可以看出，在我们设置作业的输入数据路径时，JobTracker要访问NameNode，却没有成功。

接着，我们通过查看正在运行的进程，确定了JobTracker运行正常，MapReduce作业失败的原因是无法访问NameNode。

最后，我们重启了NameNode并确认HDFS运转正常。

1. 启动替代NameNode

截至目前，我们认识到MapReduce集群和HDFS集群存在某些区别。有了这样的认识，就不会对“启用替代NameNode要比启用替代JobTracker的步骤更为复杂”感到吃惊。毫不客气地说，由于硬件故障而不得不启用替代NameNode，这可能是Hadoop集群中最糟糕的情况。除非你有充分的准备，否则极可能丢失全部数据。

这仅是一个结论声明，只有深入研究NameNode的本质，才能理解为什么会出现上述情况。

2. 详解NameNode

目前，我们学到了NameNode是DataNode进程之间的协调者，同时，它还负责使某些配置参数立即生效，如数据块复制因子。这些任务都很重要，但它们都只停留在操作层面。NameNode还负责管理HDFS文件系统的元数据，可以把它类比为传统文件系统中的文件分配表。

3. 文件系统、文件、数据块和节点

在访问文件系统时，用户通常不会关注数据块。用户要访问的是文件系统中某个位置的特定文件。为了便于实现这个功能，NameNode进程需要维护许多信息。

- 文件系统的实际内容，所有文件的文件名，以及它们所在的目录。
- 关于上述元素的元数据，比如文件大小、所有者、复制因子。
- 数据块与文件之间的映射关系，反映了哪些数据块保存的是哪个文件的数据。
- 集群中节点与数据块的映射关系，反映了哪些数据块存储在哪个节点上。此外，基于上

述映射关系，NameNode还记录了每块数据的当前副本状态。

除最后一点外，所有信息都是永久数据，必须在NameNode重启过程中维护这些数据。

4. 集群中最重要的数据：fsimage

NameNode进程在硬盘上保存了两个数据结构，一个是`fsimage`文件，另一个是edit log，它记录了`fsimage`文件的所有改动。`fsimage`文件存储了上节提到的文件系统的关键属性：每个文件及目录的文件名和详细信息，以及每个文件、目录与数据块之间的映射关系。

假如丢了`fsimage`文件，保存数据块的节点就无法获知哪些数据块对应着哪个文件的哪一部分。实际上，你甚至都不知道应该首先构建哪个文件。`fsimage`文件的丢失不会影响文件系统数据的完整性，然而，这些数据却变得毫无用处。

NameNode进程启动时会读取`fsimage`文件。为了高效运行，这些数据保存在NameNode的内存中，并在内存中完成操作。为了防止丢失对文件系统的更改，这些更改在NameNode正常运转时间被写入edit log文件。重新启动的时候，启动之时就搜寻这个日志文件，并把它的内容更新到`fsimage`文件中，之后NameNode会把`fsimage`文件的内容读入内存。



该过程可通过使用稍后将提到的SecondaryNameNode进行优化。

5. DataNode的启动

当DataNode进程启动时，它开始向NameNode进程发送心跳信息，报告存储在该节点的数据块的状态。本章前面内容曾解释过，当客户端提出对特定数据块的读写请求时，NameNode进程通过上述方法可获知由哪个节点向该客户端提供服务。如果重启了NameNode，它将与所有DataNode进程重新建立心跳机制，来构建数据块与存储节点的映射关系。

DataNode因故障离开集群，故障恢复后又重新加入集群，导致数据块与节点的映射关系无法长期保存，因为在目前的实际情况下，保存在硬盘上的状态信息可能并不是最新的。这也是NameNode进程不留存数据块与节点之间映射关系的原因。

6. 安全模式

如果用户在HDFS集群启动后不久即查看HDFS网页用户接口或者`dfsadmin`的输出，你会发现集群正处于安全模式（safe mode），同时也会看到集群离开安全模式需要达到的数据块阈值。这是数据块报告机制在发挥作用。

作为一种附加的保护措施，NameNode进程会将HDFS文件系统保持在只读模式下，直到它确认DataNode上报的数据块数量达到了副本阈值。通常情况下，只需所有DataNode上报其数据块

状态即可。但是，如果某些DataNode发生故障，NameNode需要安排重新复制部分数据块，然后集群才会达到离开安全模式的条件。

7. SecondaryNameNode

SecondaryNameNode是Haoop中命名最糟糕的实体。当读者第一次学习关键的`fsimage`文件时，可能会认为这个名为SecondaryNameNode有助于防范故障。它会不会像其名字那样，是运行在另一台主机上的NameNode的副本，当主节点发生故障时，它会接管工作，控制集群？很遗憾，答案是否定的。SecondaryNameNode的作用很特殊，它周期性读取`fsimage`和edit log，并把edit log中记录的对`fsimage`的改动应用到`fsimage`文件中，输出一个经过更新的`fsimage`文件。该方案为NameNode节省了大量启动时间。一个已长时间运行的NameNode进程，会生成一个巨大的edit log，将该文件中的所有改动一次性应用到存储在硬盘上的`fsimage`文件会花费很长时间，很容易就能浪费掉几个小时。使用SecondaryNameNode可以帮助NameNode较快启动。

8. 如何处理NameNode进程的致命故障

显然，当NameNode发生致命故障时，我们安慰自己“不要慌乱”无济于事，必须要有解决方法。有多种原因可能造成NameNode故障。这个话题太重要了，以至于我们在下一章专门用一节内容来讨论它。但现在，主要的防范措施就是配置NameNode，让它把`fsimage`和edit log输出到多个不同位置。通常，我们可以添加一个网络文件系统作为`fsimage`的输出位置，保证在NameNode主机之外还存有一个`fsimage`副本。

但是，将NameNode移到一台新主机需要手工操作，在完成这些操作的过程中，整个Hadoop集群完全失效。你需要掌握这些操作。对了，你曾在测试环境中成功完成过这些步骤！千万不要在业务集群死机，CEO朝你大喊大叫，公司发生经济损失的时候才去学习如何将NameNode移到新主机上。

9. BackupNode/CheckpointNode和NameNode HA

0.22版的Hadoop用**BackupNode**和**CheckpointNode**这两个新组件来代替SecondaryNameNode。实际上，CheckpointNode是对SecondaryNameNode的重命名，它负责在固定的checkpoint周期性地对`fsimage`文件进行更新，以减少NameNode的启动时间。

而BackupNode更像是对NameNode完整功能的热备份。它在内存中保存了主NameNode的同步状态，并从NameNode接收文件系统的更新数据流，因此，任何时刻其内存状态都是最新的。如果NameNode死机了，BackupNode更适合当新NameNode。使用BackupNode作为新NameNode的过程不是自动的，它需要手工干预，重启集群，但它能解决NameNode故障带来的麻烦。

请记住，Hadoop 1.0是0.20版本的延续，因此它不具备上述特性。

Hadoop 2.0会将上述特性扩展，最终目标是实现一种全自动的NameNode故障恢复机制，无

需人工干预从主NameNode到备份NameNode的迁移过程。**HA**（High Availability）是最早提出的一种实现上述目标的Hadoop架构改革方案，一旦实现，它将发挥重要作用。

10. 硬件故障

前几节，我们故意制造了Hadoop各组件的故障。大多数情况下，我们是用结束Hadoop进程的方式模拟主机的物理硬件故障。根据经验，很少遇到Hadoop进程发生故障，而作为集群基础的主机硬件却没有任何问题的情况。

11. 主机故障

主机故障是要考虑的最简单的情况。主机故障可以由关键硬件的问题导致，比如CPU故障、电压过低、风扇被卡住等。它会导致运行在该主机的Hadoop进程突然中止。系统软件的重要缺陷，如内核错误、I/O死锁等，也会造成同样后果。

一般来讲，假如故障导致主机崩溃、重启或其他情况使其在一段时间内无法访问，我们认为这些故障对Hadoop的影响和本章前几节的描述完全相同。

12. 主机错误

更隐蔽的问题是，主机看上去运行正常，实际上输出的是错误结果。例如，内存故障导致的数据损坏或磁盘扇区损坏，造成硬盘上的数据被破坏。

对HDFS而言，这相当于我们之前讨论的数据块丢失的情况。

在MapReduce中，没有等价机制。TaskTracker和其他大多数软件一样，它依赖于主机对数据的正确读写，而在任务运行或shuffle阶段没有相应的检测错误数据的机制。

13. 关联故障的风险

某些情况下，一个故障的起因也会造成后续多个故障，并会大大增加数据丢失的风险。但很多人在遭受损失之前从未认真考虑过这个问题。

举个例子，我曾使用由4台联网设备组成的系统工作。其中1台设备出现了故障，没人去关注这个问题。毕竟还有3台设备可以正常工作。直到它们在18小时内全部出现故障，才有人去调查。后来发现，这些设备使用的同一生产批次的硬盘存在质量问题。

问题并非总是这么匪夷所思。最常见的情况是，共享服务或共享设备出现故障。网络交换机会坏掉，电源功率会突增，空调会罢工，设备架会引起短路。下一章我们会看到，Hadoop并不是随机分配数据块的存储位置，它努力实现一种数据布局策略，尽量降低共享服务引发故障的可能性以及故障造成的损失。

一般情况下，我们讨论的上述情况不太可能出现。绝大多数情况下，出现故障的主机只是个

别现象，它并非故障危机的冰山一角。但是，请记住，千万不要轻视这些不可能出现的情况，尤其是在集群规模日益增长的时候。

由软件造成的任务失败

如前所述，实际上很少遇到Hadoop进程自己崩溃或其他组件自发失效的情况。实践中更常见的是由任务引发的故障，也就是正在集群上执行的map或reduce任务引发的故障。

缓慢运行的任务引发的故障

首先看一下如果任务暂停，或者在Hadoop看来任务停止运行将会引发的后果。

6.8　实践环节：引发任务故障

我们将会造成任务失败。在开始行动之前，我们需要修改默认的超时阈值。

(1) 将下列配置属性添加到mapred-site.xml文件。

```
<property>
<name>mapred.task.timeout</name>
<value>30000</value>
</property>
```

(2) 我们现在修改第3章中曾多次用到的WordCount例程。复制WordCount3.java，重命名为WordCountTimeout.java，并添加下列引用声明。

```
import java.util.concurrent.TimeUnit ;
import org.apache.hadoop.fs.FileSystem ;
import org.apache.hadoop.fs.FSDataOutputStream ;
```

(3) 使用下列代码替换map方法。

```
    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
String lockfile = "/user/hadoop/hdfs.lock" ;
  Configuration config = new Configuration() ;
FileSystem hdfs = FileSystem.get(config) ;
Path path = new Path(lockfile) ;
if (!hdfs.exists(path))
{
byte[] bytes = "A lockfile".getBytes() ;
   FSDataOutputStream out = hdfs.create(path) ;
out.write(bytes, 0, bytes.length);
out.close() ;
TimeUnit.SECONDS.sleep(100) ;
}
```

```
        String[] words = value.toString().split(" ") ;

        for (String str: words)
        {
                word.set(str);
                context.write(word, one);

            }
            }
        }
```

(4) 修改类名，之后编译WordCountTimeout.java并打包为jar文件，在集群上运行。

```
$ Hadoop jar wc.jar WordCountTimeout test.txt output
...
11/12/11 19:19:51 INFO mapred.JobClient:    map 50% reduce 0%
11/12/11 19:20:25 INFO mapred.JobClient:    map 0% reduce 0%
11/12/11 19:20:27 INFO mapred.JobClient: Task Id : attempt_2011121
11821_0004_m_000000_0, Status : FAILED
Task attempt_201112111821_0004_m_000000_0 failed to report status
for 32 seconds. Killing!
11/12/11 19:20:31 INFO mapred.JobClient:    map 100% reduce 0%
11/12/11 19:20:43 INFO mapred.JobClient:    map 100% reduce 100%
11/12/11 19:20:45 INFO mapred.JobClient: Job complete:
job_201112111821_0004
11/12/11 19:20:45 INFO mapred.JobClient: Counters: 18
11/12/11 19:20:45 INFO mapred.JobClient:    Job Counters
...
```


原理分析

首先，我们修改了Hadoop的默认超时属性。如果任务在这段时间里处于静默状态，那么这段时间结束后，Hadoop框架会强制结束该任务。

接着，我们对WordCount3进行修改，加入一些代码让该任务休眠100秒。程序中用到了HDFS上的文件锁，以确保只有一个任务实例处于休眠状态。如果我们只在map操作中加入休眠声明而不加上文件锁，每个mapper都会超时最终导致作业失败。

一展身手：编写代码访问HDFS

我们讲过，本书不会介绍如何编写程序访问HDFS。然而，注意一下上例中的代码，并在Java文档中查阅用到的这些类。你会发现，很大程度上，这些接口与访问标准Java文件系统的模式类似。

然后我们编译源代码，打包类文件并在集群上执行作业。第一个任务进入休眠状态，在超过

设置的超时阈值（该值以毫秒为单位）之后，Hadoop杀死该任务，并重新安排另一个mapper处理该任务负责处理的split。

1. Hadoop对运行缓慢的任务的处理方式

面对运行缓慢的任务，Hadoop有一种平衡做法。它要强制结束那些卡住的任务，或者由于其他原因运行非常缓慢的任务。但有些时候，复杂任务通常需要花费较长时间。尤其是依赖其他外部资源完成自身运行的任务，往往需要较长的运行时间。

Hadoop从任务进度中寻找线索，以推断其处于空闲状态、静默状态或卡住状态的时间。通常来讲，线索来源包括：

- ❑ 输出结果；
- ❑ 向计数器写入数值；
- ❑ 明确地报告进度。

对于后者，Hadoop提供了`Progressable`接口，该接口包含一个有趣的方法。

```
Public void progress() ;
```

`Context`类实现了`Progressable`接口，因此任意mapper或者reducer都可以调用`context.progress()`报告作业执行的进度。

2. 预测执行

通常，一个MapReduce作业包括多个离散的map和reduce任务。在集群上运行作业时，由于某台主机配置错误或者发生故障导致该主机运行的任务明显落后于其他任务的风险确实存在。

为了解决这一问题，Hadoop会在map或reduce阶段即将结束之际，在集群中的多台主机上运行相同的map或reduce任务。预测任务执行的目的在于，防止因一两个运行缓慢的任务对整个作业的运行时间产生重大影响。

3. Hadoop对失败任务的处理方法

运行失败的任务并非总是以暂停运行的形式存在。有时它们会明确地抛出异常、中止运行或以其他有声方式停止运行。

Hadoop的3个配置属性决定了应对任务故障的方式，它们都是在`mapred-site.xml`文件中进行设置。

- ❑ `mapred.map.max.attempts`：在引起作业失败之前，每个map作业的最大重试次数。
- ❑ `mapred.reduce.max.attempts`：在引起作业失败之前，每个reduce作业的最大重试次数。
- ❑ `mapred.max.tracker.failures`：如果失败的任务数超过该值，整个作业失败。

上述属性的默认值都是4.



> 请注意，如果mapred.tracker.max.failures的值小于其他两个属性中的任何一个，该设置都不会生效。
>
> 配置这些属性中的哪一个，取决于数据和作业的性质。如果作业要访问的外部资源可能经常出现暂时错误，最好增大任务的重试次数。但如果任务处理的是特定数据，这些属性可能都不太适用，因为失败过一次的任务依然还会失败。无论如何，大于1的默认值是有意义的，因为在大型复杂系统中，总会出现各种短暂故障。

一展身手：引发任务失败

再次修改WordCount程序。这次不是让任务休眠，而是基于随机数抛出RuntimeException。修改集群配置并研究多少个失败的任务会导致整个作业失败。

6.9 数据原因造成的任务故障

我们将要讨论的最后一类故障是由数据引发的。在这里，我们指的是由包含错误数据的记录导致的任务故障，可能是因为数据类型或者格式错误，也可能是其他种种相关问题。这些情况下，任务接收的数据往往背离了我们的预期。

1. 通过编写代码处理异常数据

处理异常数据的一种方法是，在mapper和reducer中实现防范异常数据的机制。例如，假如mapper接收的数据应该是一系列以逗号分隔的值，那么在处理数据之前首先验证数据数目是否正确。如果第一个值应该是一个整数的字符串表示，确保字符串向数字类型的转换有可靠的错误处理机制和默认行为。

这种方法的问题是，无论你有多细心，总会有一些怪异的输入数据类型没有考虑到。你是否考虑接收不同unicode字符集的值？遇到多个字符集、空值、错误结尾的字符串、错误编码的转义字符等情况，该怎么办呢？

如果作业的输入数据是由用户生成的，或者受用户控制，可以较少关注这些可能性。然而，如果用户处理的数据来自外部数据源，就会发生一些意想不到的情况。

2. 使用Hadoop的skip模式

另一种方法是配置Hadoop，让它以其他方式处理任务故障。与将任务故障视为原子事件不

同，Hadoop试图识别出引发问题的数据，并在将来的任务执行中跳过这些数据。这种机制被称为**skip**模式。假如你遇到各种各样的数据问题，而又不想编码解决这些问题或者编码解决这些问题是不切实际的，这个时候就用到skip模式了。又或者，你的作业使用了第三方库，而又没有这些库的源代码，可能只能使用skip模式来解决数据问题了。

出于其他考虑，skip模式目前仅适用于0.20之前版本的API编写的作业。

6.10　实践环节：使用 skip 模式处理异常数据

我们通过实际编写一个MapReduce作业来学习skip模式，该作业接收的数据会导致其运行失败。

(1) 将下列Ruby脚本保存为`gendata.rb`文件。

```
File.open("skipdata.txt", "w") do |file|
3.times do
    500000.times{file.write("A valid record\n")}
    5.times{file.write("skiptext\n")}
  end
  500000.times{file.write("A valid record\n")}
End
```

(2) 运行脚本。

```
$ ruby gendata.rb
```

(3) 检查生成文件的大小及其行数。

```
$ ls -lh skipdata.txt
-rw-rw-r-- 1 hadoop hadoop 29M 2011-12-17 01:53 skipdata.txt
~$ cat skipdata.txt | wc -l
2000015
```

(4) 将生成的`skipdata.txt`数据拷贝到HDFS。

```
$ hadoop fs -put skipdata.txt skipdata.txt
```

(5) 在`mapred-site.xml`文件中添加下列属性。

```
<property>
<name>mapred.skip.map.max.skip.records</name>
<value5</value>
</property>
```

(6) 检查`mapred.max.map.task.failures`的值，如果该值小于20的话，将其设为20。

(7) 将下列Java代码保存为`SkipData.java`文件。

```
import java.io.IOException;

import org.apache.hadoop.conf.* ;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.* ;
import org.apache.hadoop.mapred.* ;
import org.apache.hadoop.mapred.lib.* ;

public class SkipData
{

    public static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, LongWritable>
    {

        private final static LongWritable one = new
LongWritable(1);
        private Text word = new Text("totalcount");

        public void map(LongWritable key, Text value,
            OutputCollector<Text, LongWritable> output,
                Reporter reporter) throws IOException
                {
                    String line = value.toString();

                if (line.equals("skiptext"))
                throw new RuntimeException("Found skiptext") ;
                output.collect(word, one);
            }
        }

        public static void main(String[] args) throws Exception
        {
            Configuration config = new Configuration() ;
            JobConf conf = new JobConf(config, SkipData.class);
            conf.setJobName("SkipData");

            conf.setOutputKeyClass(Text.class);
            conf.setOutputValueClass(LongWritable.class);

            conf.setMapperClass(MapClass.class);
            conf.setCombinerClass(LongSumReducer.class);
            conf.setReducerClass(LongSumReducer.class);

            FileInputFormat.setInputPaths(conf,args[0]) ;
            FileOutputFormat.setOutputPath(conf, new
Path(args[1])) ;

            JobClient.runJob(conf);
        }
    }
```

(8) 编译该文件，并将其打包为skipdata.jar。

(9) 运行作业。

```
$ hadoop jar skip.jar SkipData skipdata.txt output
...
11/12/16 17:59:07 INFO mapred.JobClient:  map 45% reduce 8%
11/12/16 17:59:08 INFO mapred.JobClient: Task Id : attempt_2011121
61623_0014_m_000003_0, Status : FAILED
java.lang.RuntimeException: Found skiptext
  at SkipData$MapClass.map(SkipData.java:26)
  at SkipData$MapClass.map(SkipData.java:12)
  at org.apache.hadoop.mapred.MapRunner.run(MapRunner.java:50)
  at org.apache.hadoop.mapred.MapTask.runOldMapper(MapTask. java:358)
  at org.apache.hadoop.mapred.MapTask.run(MapTask.java:307)
  at org.apache.hadoop.mapred.Child.main(Child.java:170)


11/12/16 17:59:11 INFO mapred.JobClient:    map 42% reduce 8%
...
11/12/16 18:01:26 INFO mapred.JobClient:    map 70% reduce 16%
11/12/16 18:01:35 INFO mapred.JobClient:    map 71% reduce 16%
11/12/16 18:01:43 INFO mapred.JobClient: Task Id : attempt_2011111
61623_0014_m_000003_2, Status : FAILED
java.lang.RuntimeException: Found skiptext
...
11/12/16 18:12:44 INFO mapred.JobClient:    map 99% reduce 29%
11/12/16 18:12:50 INFO mapred.JobClient:    map 100% reduce 29%


11/12/16 18:13:00 INFO mapred.JobClient:    map 100% reduce 100%
11/12/16 18:13:02 INFO mapred.JobClient: Job complete:
job_201112161623_0014
...
```

(10) 检查作业输出文件的内容。

```
$ hadoop fs -cat output/part-00000
totalcount    2000000
```

(11) 在输出路径中查看跳过的数据。

```
$ hadoop fs -ls output/_logs/skip
Found 15 items
-rw-r--r--    3 hadoop supergroup    203 2011-12-16 18:05 /
user/hadoop/output/_logs/skip/attempt_201112161623_0014_m_000001_3
-rw-r--r--    3 hadoop supergroup    211 2011-12-16 18:06 /
user/hadoop/output/_logs/skip/attempt_201112161623_0014_m_000001_4
...
```

(12) 通过MapReduce UI查看作业详情，观察统计数据，如下图所示。

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	8	0	0	8	0	22 / 1
reduce	100.00%	1	0	0	1	0	0 / 0

	Counter	Map	Reduce	Total
Job Counters	Launched reduce tasks	0	0	1
	Rack-local map tasks	0	0	16
	Launched map tasks	0	0	31
	Data-local map tasks	0	0	15
SkippingTaskCounters	MapProcessedRecords	2,000,000	0	2,000,000
FileSystemCounters	FILE_BYTES_READ	378	321	699
	HDFS_BYTES_READ	30,028,814	0	30,028,814
	FILE_BYTES_WRITTEN	997	321	1,318
	HDFS_BYTES_WRITTEN	670	19	689
	Reduce input groups	0	1	1
	Map skipped records	15	0	15
	Combine output records	15	0	15

原理分析

上例中，我们完成了许多设置，现在将对其进行逐条解释。

首先，我们需要对Hadoop进行配置，开启skip模式，默认情况下，该模式处于关闭状态。关键是将mapred.skip.map.max.skip.records的值设置为5，这就是说，我们指令Hadoop框架跳过小于5条记录的数据集。请注意，这个值包括无效记录的数量，如果将该属性的值设为0（默认设置），Hadoop就不会进入skip模式。

我们也检查mapred.max.map.task.failures的值，确保作业会重试足够多次，具体原因稍后解释。

接下来，我们需要使用一个测试文件模拟异常数据。我们编写了一个简单的Ruby脚本生成一个文件，该文件包含2 000 000行有效数据，以及分散在文件中的3组无效数据，每组包括5条无效记录。我们运行该脚本，并确认生成的文件确实包括2 000 015行。然后，将该文件放到HDFS上。

随后，我们编写一个简单的MapReduce作业统计有效记录数量。作业每次从输入文件读取一

行，如果该行为有效记录，输出值1，最终这些输出值聚合成为最终输出值。当遇到无效数据行时，mapper抛出一个异常并失败。

接下来编译上述作业源代码，打包为jar文件，并运行作业。一段时间后，作业运行结束。从作业状态摘要中可以看出，我们从未见过这种运行模式。随着map任务的运行，map进度计数器的值不断增大，但当任务失败时，map任务进度后退然后再次增大。这就是skip模式。

每当mapper接收到键值对时，Hadoop默认增加计数器的值，这样就会记录哪条数据引发了故障。



如果map或reduce任务并非直接通过map或reduce方法的参数来接收输入数据，（例如，从异步进程或缓存中接收数据）你就需要手动更新此计数器。

任务失败时，Hadoop在相同数据块上重试，但只试图解决无效的记录。通过二分搜索方法，Hadoop框架在数据上重试，直到跳过的记录数小于我们之前设置的最大值（本例中该值为5）。因为Hadoop框架要找的是跳过记录的最佳值，因此这个过程会导致多次任务重试和失败，这也是我们需要将任务重试次数设置为略大于通常值的原因。

我们等待作业持续进行这种往复处理，完毕后查看输出文件的内容。该文件包含2 000 000条已处理记录，正是输入文件中的有效记录数。Hadoop成功地跳过了3组无效记录。

接着，我们检查作业输出路径下的`_logs`目录，发现该目录下有一个skip目录，存放着被跳过记录组成的序列文件。

最后，通过MapReduce网页用户接口查看整个作业的状态，既包括在skip模式下处理的记录数，也包括跳过的记录数。请注意，失败的任务总数为22。但这个数字是多个任务失败次数的总和，因此即使大于我们设置的map任务重试次数也是合理的。

是否开启skip模式

skip模式可有效解决错误数据带来的问题。但正如我们刚才看到的，因为Hadoop需要确定跳过哪些范围内的数据，这就带来了性能损失。在我们的测试文件中，无效记录刚好被分为3组，它们只占全部数据集的很小一部分，利于Hadoop在skip模式下运行。但如果输入数据包含许多无效记录，并且它们散布于文件中，更有效的办法可能是使用预处理Mapreduce作业滤掉所有的无效记录。

这也是我们在介绍了编写代码处理错误数据的方法之后，紧接着就介绍skip模式的原因。它们都是你应当掌握的有效技术。不存在什么情况下哪种方法更好的问题，读者需要综合考虑输入数据、性能需求以及是否具备编写处理代码的条件等因素，然后决定采用哪种办法。

6.11 小结

本章中，我们人为制造了许多破坏。希望你永远不会在同一天遇到如此多Hadoop集群故障。通过学习处理这些故障，你会掌握一些关键技术。

通常，不必害怕Hadoop的组件发生故障。尤其是在大规模集群中，一些组件或主机发生故障是司空见惯的事，Hadoop在设计之初就考虑到了这种情况并提出了相应的处理方法。HDFS不仅负责存储数据，还管理着每个数据块的副本。在DataNode进程意外结束时，HDFS会安排生成新的副本。

MapReduce作业是无状态的。如果某个TaskTracker执行的任务发生故障，通常只需在另外的节点上重新执行相同任务。这个方法也可用于防止发生故障的主机拖慢整个作业的进度。

HDFS和MapReduce主节点的故障更为严重。特别是，NameNode进程保存着文件系统的关键数据，必须保证在它发生故障时有一个新NameNode进程接班。

通常来说，硬件故障看上去与前面所说的故障比较接近，但要留意发生关联故障的可能性。如果软件错误导致任务故障，Hadoop会按照设置限值重置多次。由数据问题引发的错误可以通过采用skip模式解决，尽管它会带来性能损失。

现在我们已学习了如何处理集群故障，下一章我们将介绍集群设置、健康状况和集群维护方面可能遇到的问题。

第7章 系统运行与维护

我们不能光是在Hadoop集群上运行写好的程序进行智能数据分析，同时也得负责维护集群，做好数据处理的准备工作。

本章包括以下内容：

- 讨论更多的Hadoop配置属性；
- 如何挑选集群硬件；
- Hadoop安全机制的工作原理；
- 管理NameNode；
- 管理HDFS；
- 管理MapReduce；
- 扩展集群规模。

尽管对上述话题的讨论都只停留在操作层面，但是我们依然可以通过学习这些内容，研究一些从未尝试的Hadoop功能。因此，即使读者并不需要亲自管理集群，这些信息也是有用的。

7.1 关于 EMR 的说明

使用Amazon Web Services之类的云服务的好处之一是，云服务提供者负责大部分系统维护工作。弹性MapReduce既可以创建执行单个任务的Hadoop集群（非永久作业流），也可以创建长期运行的执行多个作业的集群（永久作业流）。当使用非永久作业流时，在很大程度上，底层Hadoop集群的配置和运行方式对用户是不可见的。因此，使用非永久作业流的用户不需要考虑本章的很多话题。如果用户使用EMR执行永久作业流，那么就和本章的多个话题（并非全部话题）相关。

一般来讲，本章讨论的是本地Hadoop集群。如果用户需要重新配置永久作业流，按照第3章的内容设置相同的Hadoop属性即可。

7.2 Hadoop 配置属性

在运行集群之前，我们来讨论一下Hadoop的配置属性。一直以来，我们介绍了很多Hadoop的配置属性，但还有一些其他内容值得深入思考。

默认值

让Hadoop新用户感到最为困惑的就是配置属性太多。它们包含在哪个文件里面？它们是什么意思？它们的默认值是什么？

如果你使用的是Hadoop的完整版本，或者说不仅仅是二进制版本，下面这些XML文件会回答你的问题。

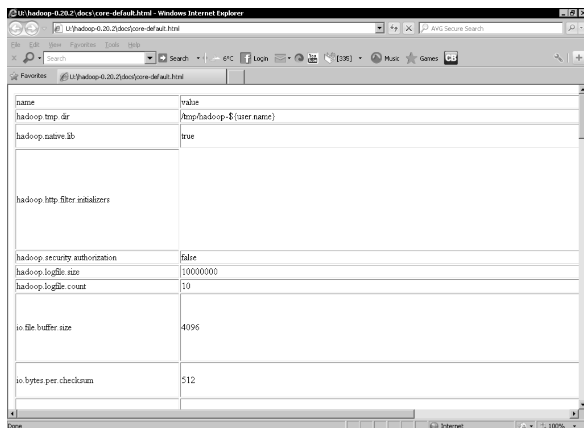
- `Hadoop/src/core/core-default.xml`
- `Hadoop/src/hdfs/hdfs-default.xml`
- `Hadoop/src/mapred/mapred-default.xml`

7.3 实践环节：浏览默认属性

幸运的是，XML 文档不是查看这些默认值的唯一选择，我们还可以选择可读性更强的 HTML 版本。下面我们将快速浏览 HTML 版的默认配置。

Hadoop 二进制版中不包含这些 HTML 文件。如果读者使用的正是二进制版的 Hadoop 安装包，也可以在 Hadoop 网站上找到这些文件。

(1) 使用浏览器打开 Hadoop 安装路径下的 `docs/core-default.html` 文件，并浏览其内容。相应的页面如下图所示。



(2) 通过类似方法浏览其他两个文件。

- ❑ Hadoop/docs/hdfs-default.html
- ❑ Hadoop/docs/mapred-default.html

原理分析

如你所见，每个属性都包含名称、默认值及简要描述。还会看到，确实有很多配置属性。现在别想着能理解所有属性的含义，但要花点时间了解Hadoop支持的自定义类型。

7.3.1　附加的属性元素

刚才，我们设置配置文件中的属性时用到了XML元素，它们的格式如下所示。

```
<property>
<name>the.property.name</name>
<value>The property value</value>
</property>
```

我们可以增加两个可选的XML元素，它们是description和final。使用了上述两个附加元素的完整属性应如下所示。

```
<property>
<name>the.property.name</name>
<value>The default property value</value>
<description>A textual description of the property</description>
<final>Boolean</final>
</property>
```

description元素很明显，无需太多解释，使用它可以为HTML文件中的每个属性提供描述性文本。

final元素的含义与它在Java中的含义类似：用户无法使用其他文件中指定的属性值改写包含final元素的属性的值，也没有其他方法可以覆盖该属性。稍后我们会看到这样的例子。在关系到集群性能、完整性、安全性或由于其他原因，用户希望在整个集群中使用同一个属性值的情况下，可以在这些属性中使用final元素。

7.3.2　默认存储位置

用户可以通过配置存储位置这一属性，改变Hadoop在本地硬盘和HDFS上的数据存储位置。

hadoop.tmp.dir是其他属性的基础，它是所有Hadoop文件的根目录，其默认值为/tmp。

遗憾的是，包括Ubuntu在内的许多Linux版本在每次重启时都会清空该目录的内容。这就意味着，如果用户不修改这个属性的值，就会在下次重启时丢失所有HDFS数据。因此，有必要像下面这样改写core-site.xml文件中的hadoop.tmp.dir属性的值。

```
<property>
<name>hadoop.tmp.dir</name>
<value>/var/lib/hadoop</value>
</property>
```

别忘了，要确保Hadoop用户对该位置具有写入权限。同时，该目录所在的硬盘有充足空间。稍后你会看到，有许多其他属性可以更细粒度地控制特定类型数据的存储位置。

7.3.3 设置Hadoop属性的几种方式

刚才，我们使用了配置文件对Hadoop的属性值进行设置。这是一种设置Hadoop配置属性的有效途径，但如果我们想试着找出某个属性的最佳值，或者需要在执行某一作业的时候设置特殊值，这种方法就显得有点繁琐。

我们可以使用JobConf类编写程序为正在运行的作业设置配置属性。该类支持两种函数：一类函数用于设置某个特定属性的值，比如我们曾看到的设置作业名称、输入输出格式及其他属性的值。另一种函数用来设置作业的map和reduce任务数等。

此外，还有一些通用方法，如下所示。

- Void set(String key, String value);
- Void setIfUnset(String key, String value);
- Void setBoolean(String key, Boolean value);
- Void setInt(String key, int value);

这些函数的使用更为灵活，用户无需为每个要修改的属性都创建一个专用方法。然而，这些函数都缺少编译时检查机制。这就意味着，如果使用一个无效的属性名或为某个属性指定了错误类型，可能直到函数运行时才会发现这些问题。



用户既可以编写程序来设置属性值，也可以在配置文件中设置属性值。这就是为配置属性设计final元素的一个重要原因。假如用户不希望某些属性的值被提交的MapReduce作业覆盖，就需要在主配置文件中为这些属性添加final元素。

7.4　集群设置

在学习如何维护集群，使其保持运转之前，我们需要首先研究如何设置集群。

7.4.1　为集群配备多少台主机

在考虑Hadoop新集群时，第一个问题就是首次为该集群配备多少台主机。我们知道，随着业务所需计算能力的增长，可以在集群中新增节点，但我们也想在设计之初即分配合适的主机数，免得马上就需要增加新节点。

这个问题还真是没有明确的答案，它在很大程度上依赖于要处理的数据集规模和要执行的作业的复杂程度。我们只能近似地说，节点数至少要与复制因子*n*持平。请记住，节点是会发生故障的，如果集群中的节点数等于复制因子，那么任意一个节点故障都会造成数据块的副本数低于复制因子。在大部分由数十个或数百个节点组成的集群中，这个问题不足为虑。但在很小的集群中，如果复制因子是3，最安全的方案是用5台主机构建集群。

1. 计算节点的可用空间

确定集群所需节点的直观方法是，评估要用该集群处理的数据集的规模。如果要处理10 TB数据，并且主机硬盘空间为2 TB，结论就是至少需要5台主机。

这是不对的，因为它没有把复制因子和临时空间的因素考虑进去。回想一下，mapper的输出被写到本地硬盘，再被reducer读取。我们需要把这个较大的硬盘使用率考虑进去。

一种好的做法是，假设复制因子为3，同时临时空间要占用25%的硬盘原始空间。基于上述假设，要在主机硬盘空间为2 TB的集群上处理10 TB数据，所需主机数的计算方法如下。

- 用主机存储空间总量除以复制因子。

  2 TB/3 = 666 GB
- 在此基础上减去25%的临时数据存储空间。

  666 GB * 0.75 = 500 GB
- 因此，每个硬盘存储空间为2 TB的节点只有大约500 GB（0.5 TB）的可用空间。
- 数据集规模除以该值，结果即为所需的节点数。

  10 TB / 500 GB = 20

所以，处理10 TB数据的集群最少需要20个节点，它是我们初步估算值的4倍。

所需节点数多于预期，这种情况很普遍。在考虑主机规格的时候要记住这个前提，本章“硬件选型”会讲到这个问题。

2. 主节点的位置

下一个问题是，在哪台主机上运行NameNode、JobTracker和SecondaryNameNode。我们曾看到过，DataNode可以与NameNode运行在同一台主机上，TaskTracker也可以和JobTracker共同运行在一台主机上，但这并不是产品集群的主流设置。

我们将看到，NameNode和SecondaryNameNode有一些特殊的资源需求，所有可能影响这两个主节点性能的因素都可能拖慢整个集群的运算性能。

最理想的情况是，在专用主机上运行NameNode、JobTracker和SecondaryNameNode。然而，在规模很小的集群中，这将明显增加硬件成本，却不一定能够从中充分受益。

如果可能的话，首先应当在一台专用主机上运行NameNode、JobTracker和SecondaryNameNode，不要在这台专用主机上再运行任何DataNode或者TaskTracker进程。随着集群规模增长，可以在集群中新增一台服务器主机，并把NameNode迁移到该主机，让JobTracker和SecondaryNameNode运行在同一台主机上。最后，随着集群规模的进一步增长，有必要再把JobTracker和SecondaryNameNode分开，分别在单独的主机上运行。



就像在第6章中所讨论的，Hadoop 2.0会把Secondary NameNode分为Backup NameNode和Checkpoint NameNode。最好分别在不同的专用主机上运行Backup NameNode和Checkpoint NameNode，但在条件不足的情况下，至少要保证Backup NameNode运行在一台专用主机上。

3. 硬件选型

在确定节点所用硬件规格时，不能只考虑要存储的数据量大小。除此之外，还要考虑节点的处理能力、内存大小、存储类型以及网络带宽。

我们讨论了很多为Hadoop集群选用硬件的内容，再次声明，没有一个答案适用于各种情况。MapReduce作业的类型在硬件选型中起着决定性作用，尤其是，这些作业是否受限于CPU、内存、I/O或其他硬件的性能。

4. 处理器/内存/硬盘空间比

考虑作业是否受限于处理器、内存或者I/O的一个好办法是检查CPU/内存/硬盘空间比。例如，在考虑CPU/内存/硬盘空间比的情况下，我们认为一台有着8 GB内存、2 TB硬盘的四核主机相当于一台内存大小为4 GB、硬盘大小为1 TB的双核主机。

然后，检查一下将要运行的MapReduce作业的类型，这样的比率是否合适？换句话说，数据处理任务更依赖于哪种资源？是否需要一种比较平衡的配置？

当然，最好通过原型设计和各种指标来评估这个问题，但有时候这些方法却行不通。如果行不通的话，考虑一下作业的哪个阶段最耗资源。例如，我们曾见过一些I/O密集型作业，它们从硬盘读取数据，执行简单的转换之后就把结果写回硬盘。如果我们的工作任务具有这样的特点，可能需要使用存储空间更大的硬盘（尤其是在通过使用多个硬盘增加I/O的情况下时），可以相应地减少CPU和内存。

相反，执行繁重的数字运算任务的作业需要更多的CPU，那些创建或使用大型数据结构的作业则需要更大的内存。

也可以从作业的限制因素的角度考虑这个问题。运行中的作业是计算密集型（处理器满功率运作、内存和I/O过剩）、内存密集型（物理内存已满并有部分数据交换到硬盘、CPU 和I/O过剩），或I/O密集型（CPU和内存过剩，但从硬盘读写数据的速度已达到最大）中的哪一类？是否能通过增加相应的硬件来改善这些状况？

当然，用户需要不断调整并优化硬件配置，因为一旦解决了某个限制因素后，另一个原本不明显的问题就会暴露出来。所以一定要记住，整体思路是在作业应用场景中达到最佳的性能配置。

假如你确实不知道作业的性能特点，又该怎么办呢？理想情况是，基于现有硬件进行原型设计，并使用该原型系统辅助决策。但是，假如这些都无法实现，用户只能不断调整配置并通过实验来验证某种配置是否合适。请记住，Hadoop支持使用不同规格的硬件，尽管统一规格的硬件最终会简化集群的创建，所以搭建一个最小规模的集群并评估硬件是否合适。这些评估结论可以为购买新主机或升级现有硬件提供辅助信息。

5. 基于EMR进行原型设计

回想一下，在弹性MapReduce集群中配置作业时，我们会为主节点、DataNode和TaskTracker选择不同类别的硬件。假如你打算在EMR上运行作业，EMR平台提供了调整硬件配置的内置方案，可以找到既满足价格预算又符合执行速度要求的硬件。

但是，假如读者不打算在EMR上运行作业，还可以把它用作重要的原型设计平台。假如用户在为集群选配硬件却不知道作业的性能特点，可以考虑在EMR上进行原型设计以深入理解作业特点。尽管这样可能会支付一些未曾考虑过的EMR服务费用，但这些费用远远小于买了一堆完全不合适的硬件的开销。

7.4.2　特殊节点的需求

并非所有主机的硬件需求完全相同。尤其是，承载NameNode进程的主机的硬件可能与运行DataNode和TaskTracker的主机硬件完全不同。

回想一下，NameNode在内存中维护了HDFS文件系统，文件、路径、数据块、节点之间的映射关系，以及与上述内容相关的多种元数据。这就意味着，NameNode可能会受限于内存，与其他主机相比，它需要更大的内存，尤其是在集群规模很大或HDFS上存储了大量文件时，NameNode对内存的需求更为强烈。我们通常为DataNode或TaskTracker选配16 GB内存，但却需要为NameNode选配64 GB的内存，甚至更大。如果NameNode的物理内存已耗尽并开始使用虚拟内存，这将严重影响集群性能。

然而，尽管对物理内存而言，64 GB已不是一个小数字，但对现在的硬盘空间而言，这个数字还是太小。由于NameNode节点的硬盘只负责存储文件系统镜像，不需要为其配备常用于DataNode的大容量硬盘。我们更关注的是NameNode的可靠性，所以需要按照冗余配置的要求为其配备多块硬盘。因此，为了达到冗余备份的目的，与大容量硬盘相比，NameNode主机更喜欢选用多块小容量硬盘。

总的来讲，NameNode主机与集群中其余主机不同。这也是我们之前建议只要预算允许，就将NameNode迁移到专用主机上的原因，因为它的特殊硬件需求较为容易满足。



SecondaryNameNode（Hadoop2.0中的CheckpointNameNode和BackupNameNode）对硬件的需求与NameNode相同。因为它起的是备份替代作用，用户可以在更通用的主机上运行SecondaryNameNode。但如果主节点发生故障，用户需要将NameNode迁移到备用节点时，通用硬件可能会带来麻烦。

7.4.3 不同类型的存储系统

尽管用户可能在“处理器、内存、硬盘存储空间或I/O的相对重要性”这一问题上坚持自己的观点，这些争论通常都以应用程序的需求、硬件特点和指标为基础。但是，在讨论选用哪种类型的存储系统时，人们经常会因为根深蒂固的观念而发生激烈的争执。

1. 通用存储与企业级存储的对比

第一个争议是选用通用硬盘还是选用企业用户硬盘呢？哪种选择更有意义？前者（主要是SATA硬盘）体积较大，价格更便宜，读写速度相对较慢，平均无故障时间（MTBF）更短。企业级硬盘使用了SAS或光纤通道技术，总体上更为小巧，价格更高，读写更快，平均无故障时间较长。

2. 独立硬盘与RAID的对比

第二个问题是选择哪种硬盘配置方式。企业级的解决方案是使用廉价磁盘冗余阵列（RAID）把多个硬盘聚合成一个独立的逻辑存储设备。它以损失整体存储能力以及读写速率为代价，可以不动声色地解决一个或多个硬盘的故障。

另一种方案是独立使用每个硬盘，以达到总体存储能力和I/O最大化，但其缺点是，在某个硬盘发生故障的情况下，主机就会宕机。

3. 综合考虑

在许多方面，Hadoop系统会假设硬件故障不可避免。从这个角度来看，可能不需要使用传统的企业关注的存储功能。反而可以使用许多大体积的廉价硬盘增大整体存储能力，通过并行读写提高I/O吞吐率。单个硬盘的故障可能导致主机失败，但我们已学习过，集群会处理这个故障。

这是一个非常有效的方案，它在许多情况下发挥了重要作用。但是，这个方案忽视了修复故障主机的成本。如果用户使用的集群就在隔壁房间，并且手头上有许多闲置硬盘，主机修复工作就是一个不耗时的、易实现的、低成本的任务。但是，如果用户集群运行在商业托管设备上，亲自动手的维护成本相对较高。如果用户使用的是完全托管的服务器，需要向服务提供者支付维护费用，那么维护成本会更高。在这种情况下，虽然使用RAID会减少整体容量以及降低I/O吞吐率，但其能够降低维护成本，也是一种值得考虑的方案。

4. 网络存储系统

最没道理的就是使用网络存储系统作为主集群存储系统。不管是通过**SAN**（Storage Area Network，存储区域网络）技术实现的数据块存储还是通过**NFS**（Network File System，网络文件系统）或类似协议实现的基于文件的网络存储，这些方案都引入了一些不必要的瓶颈问题和可能引发故障的共享设备，从而限制了Hadoop。

但是，某些时候，由于一些非技术原因，你不得不采用类似方案。并不是说采用这些方案会导致Hadoop停止工作，而是会影响Hadoop的运行速度以及容错能力。因此，用户要能够正确理解采用这些方案带来的后果。

7.4.4　Hadoop的网络配置

与它对存储设备的支持相比，Hadoop对网络设备的支持要简单得多。因此，与CPU、内存和硬盘选择相比，用户可选的网络设备硬件少之又少。目前，Hadoop只支持一台网络设备，并且无法把一台主机上的所有4千兆以太网连接聚合成4千兆吞吐量。如果用户需要的网络吞吐量大于1千兆，除非硬件或操作系统可以把多个端口整合成Hadoop眼中的一台网络设备，否则唯一的办法就是使用10千兆以太网设备。

1. 数据块放置策略

我们讨论了很多HDFS使用副本作为冗余备份的内容，但还没有研究Hadoop如何决定放置数据块副本的位置。

在大多数传统服务器群中，各种主机（包括网络设备和其他设备）都被放置在标准尺寸的机柜上，垂直地一层一层地堆叠起来。每个机柜通常都有一个为其供电的通用配电装置，此外还有一个网络交换器用于将机柜上的主机接入到更广泛的网络。

基于这种结构，我们可以确定三大类故障类型：

- 影响单台主机的故障（例如，CPU、内存、硬盘、主板故障）；
- 影响机柜上所有主机的故障（比如，供电装置或交换机故障）；
- 影响整个集群的故障（例如，更多电源或网络故障，制冷或环境监测系统运行中断）。



请记住，Hadoop目前不支持分布于多个数据中心的集群，因此，第三类故障极有可能导致集群崩溃。

默认情况下，Hadoop按照每个节点都在同一个物理机柜上那样处理节点。这就意味着，每对主机之间的带宽和时延近似相等，并且每个节点受相关故障影响的可能性与其他节点相同。

2. 机柜认知

但是，如果用户使用了多机柜结构，或采用了与先前假设矛盾的其他配置，用户可以为每个节点提供向Hadoop报告其所在机柜ID的能力。Hadoop会在安排副本位置时考虑到这一点。

在这种设置中，Hadoop设法将某个节点的第一个副本放置在该主机上，第二个副本放置在同一个机柜中的另一台主机上，第三个副本放在其他机柜中的一台主机上。

这种策略较好地平衡了性能和可用性这两个因素。在机柜带有自己的网络交换机时，同一机柜内部主机之间的通信比它与另一机柜上主机通信的延迟要小。把两个副本放在同一个机柜内的主机上，保证了对这些副本的最大写入速度，而在机柜外放置一个副本则可在机柜出现故障的情况下提供冗余。

- 报告自身所在机柜的脚本

如果用户设置了`topology.script.file.name`属性并把它指向NameNode节点上的一个可执行脚本，NameNode会用这个脚本确定每台主机所在的机柜ID。

请注意，该属性需要用户进行设置，同时可执行脚本只能位于NameNode主机。

NameNode向该脚本传入它要调查的节点的IP地址，脚本负责实现从节点IP地址向其所在机柜名的映射。

如果没有指定可执行脚本的位置，所有节点都向Hadoop报告它们位于同一个默认机柜。

7

7.5 实践环节：查看默认的机柜配置

我们看一下如何为集群设置默认的机柜配置。

(1) 执行以下命令。

```
$ Hadoop fsck -rack
```

(2) 输出结果应类似于以下内容。

```
Default    replication factor:      3
Average    block replication:      3.3045976
Corrupt    blocks:                   0
Missing    replicas:               18 (0.5217391 %)
Number of data-nodes:          4
Number of racks:             1


The filesystem under path '/' is HEALTHY
```

原理分析

我们既对刚才用到的命令感兴趣，也对它的输出感兴趣。**hadoop fsck**工具用于检测并修复文件系统问题。可以看出，该命令包含的一些信息与我们常用的`hadoop dfsadmin`命令有些相似，尽管后者更倾向于详细报告每个节点的状态，而`hadoop fsck`则是报告整个文件系统内部构件信息。

hadoop fsck输出的一部分信息表明了集群中的机柜总数。从上例的输出可以看出，默认值为`1`。



> 上述命令是在刚刚用作HDFS故障修复测试的集群上执行的。这就是解释了为什么“average block replication”和“under-replicated blocks”的值显得有点异常。
>
> 如果主机发生临时性故障，故障修复后又重新回到Hadoop集群时，可能会导致数据块的副本数量大于复制因子。Hadoop不仅会增加副本以使数据块副本数量满足复制因子的要求，它还会删掉多余副本使数据块副本数量等于复制因子。

7.6 实践环节：报告每台主机所在机柜

我们可以通过创建一个为每台主机获取机柜位置的脚本，改进默认的机柜设置。

(1) 在 NameNode 主机的 Hadoop 用户主目录下创建一个 rack-script.sh 脚本，其内容如下。请记住将其中 IP 地址改为用户所用 HDFS 节点的 IP。

```
#!/bin/bash

if [ $1 = "10.0.0.101" ]; then
     echo -n "/rack1 "
else
     echo -n "/default-rack "
fi
```

(2) 把 rack-script.sh 脚本做成可执行文件。

```
$ chmod +x rack-script.sh
```

(3) 在 NameNode 主机的 core-site.xml 文件中加入下列属性。

```
<property>
<name>topology.script.file.name</name>
<value>/home/Hadoop/rack-script.sh</value>
</property>
```

(4) 重启 HDFS。

```
$ start-dfs.sh
```

(5) 通过 fsck 查看文件系统。

```
$ Hadoop fsck -rack
```

其输出如下图所示：

```
hadoop@headnode:~$ hadoop fsck -rack
FSCK started by hadoop from /127.0.0.1 for path / at Wed Jan 02 15:02:13 PST 201
3
.Status: HEALTHY
 Total size:    75342464 B
 Total dirs:    3
 Total files:   1
 Total blocks (validated):      18 (avg. block size 4185692 B)
 Minimally replicated blocks:   18 (100.0 %)
 Over-replicated blocks:        0 (0.0 %)
 Under-replicated blocks:       0 (0.0 %)
 Mis-replicated blocks:         0 (0.0 %)
 Default replication factor:    3
 Average block replication:     3.0
 Corrupt blocks:                0
 Missing replicas:              0 (0.0 %)
 Number of data-nodes:          3
 Number of racks:               2
FSCK ended at Wed Jan 02 15:02:13 PST 2013 in 1 milliseconds


The filesystem under path '/' is HEALTHY
hadoop@headnode:~$
```

原理分析

首先，我们创建了一个脚本文件，它为某个节点返回“rack1”，为其余节点返回默认值。我们把该脚本放到NameNode上，并将所需的配置属性加入NameNode的`core-site.xml`文件。

在重启HDFS之后，我们使用`hadoop fsck`获得文件系统信息。可以看到，集群目前被设置为有两个机柜。Hadoop得知集群中存在两个机柜后，它会采用之前介绍的更为复杂的数据块放置策略。



使用外部主机文件

一个通用的办法是使用类似于Unix的`/etc/hosts`的数据文件指定IP地址与机柜的映射关系，每行存储一个映射。用户可以独立更新该文件，然后使用rack-awareness脚本读取文件内容。

究竟什么是商用硬件

让我们回顾一下集群中主机的共有特点，它们看上去更像是日常使用的组装服务器还是专门构建的高端企业服务器?

有一部分问题在于，“日用品”是一个含糊的字眼。在某个业务中看来便宜的商品对另一个业务来讲就可能是豪华高档的。我们建议在选择硬件时考虑以下几个要素，然后再愉快地选择。

- 使用这些硬件，你是否需要为保证可靠性付出代价，比如再次实现Hadoop的一些容错机制?
- 付款购买的高端硬件的功能能否解决现实环境中遇到的需求或困难?
- 有没有验证过，是购买高端硬件更费钱，还是使用便宜的、可靠性稍差的硬件并解决由此带来的问题的花费更多?

随堂测验：创建集群

问题1　在为Hadoop新集群选择硬件时，下列哪个因素是最重要的?

(1) CPU内核数量以及它们的运算速度。
(2) 物理内存的容量。
(3) 硬盘的存储容量。
(4) 硬盘的读写速度。
(5) 很大程度上，它由工作量决定。

问题2 下列哪个原因导致你不愿意在集群中采用网络存储方案?

(1) 它可能引入一些新的单点失效问题。

(2) 它有一些冗余备份和容错方法，由于Hadoop已经提供了容错机制，这些特性是不必要的。

(3) 这种单个设备的性能不如同时使用多个本地硬盘的性能好。

(4) 上述选项都正确。

问题3 假如你需要在集群上处理10 TB数据。MapReduce主作业负责处理金融交易，用这些交易生成数据统计模型并预测未来。你将为集群选择哪种硬件配置方案?

(1) 20台配有双核处理器，4 GB内存以及500 GB硬盘的主机。

(2) 30台配有双核处理器，8 GB内存以及2块500 GB硬盘的主机。

(3) 30台配有四核处理器，8 GB内存以及1块1 TB硬盘的主机。

(4) 40台配有四核处理器，16 GB内存以及4块1 TB硬盘的主机。

7.7 集群访问控制

一旦崭新的集群安装并运行起来之后，用户需要考虑访问控制以及安全性问题。谁可以访问集群上的数据？是否有一些敏感数据不希望整个用户群都可以访问?

Hadoop安全模型

Hadoop一直都缺乏安全机制，直到最近，才出现了一种充其量只能称之为“标记”的安全模型。它把每个文件与其创建者及所属用户组关联起来，但却很少验证特定的客户端连接。强安全机制不仅会管理文件标记，而且会验证所有连接到Hadoop的用户身份。

7.8 实践环节：展示 Hadoop 的默认安全机制

我们曾展示过一些文件列表，这些文件的属性包括所属用户及用户组的名称。但是，我们未曾研究它们的真正意义。

(1) 在 hadoop 用户的根目录下创建一个测试文本文件。

```
$ echo "I can read this!" >    security-test.txt
$ hadoop fs -put security-test.txt    security-test.txt
```

(2) 更改文件权限，使其只能被文件创建者访问。

```
$ hadoop fs -chmod 700 security-test.txt
$ hadoop fs -ls
```

7

上述命令的输出结果如下图所示。

```
hadoop@headnode: ~
File  Edit  View  Terminal  Help
hadoop@headnode:~$ hadoop fs -ls
Found 2 items
-rw-------   3 hadoop supergroup         16 2013-01-02 15:14 /user/hadoop/security-test.txt
-rw-r--r--   3 hadoop supergroup   75342464 2013-01-02 14:56 /user/hadoop/ufo.tsv
hadoop@headnode:~$
```

(3) 确认你仍然可以读取该文件内容。

```
$ hadoop fs -cat security-test.txt
```

你会在屏幕上看到下面这行字。

```
I can read this!
```

(4) 连接到集群中的另一个节点并试着读取文件内容。

```
$ ssh node2
$ hadoop fs -cat security-test.txt
```

你会在屏幕上看到下面这行字。

```
I can read this!
```

(5) 从其他节点退出系统。

```
$ exit
```

(6) 为其他用户创建根目录，并赋予它们所有权。

```
$ hadoop m[Kfs -mkdir /user/garry
$ hadoop fs -chown garry /user/garry
$ hadoop fs -ls /user
```

上述命令的结果如下图所示：

```
hadoop@headnode: ~
File  Edit  View  Terminal  Help
hadoop@headnode:~$ hadoop fs -ls /user
Found 2 items
drwxr-xr-x   - garry  supergroup          0 2013-01-02 15:18 /user/garry
drwxr-xr-x   - hadoop supergroup          0 2013-01-02 15:14 /user/hadoop
hadoop@headnode:~$
```

(7) 切换到 garry 用户。

```
$ su garry
```

(8) 试着读取 hadoop 用户根目录下的测试文件。

```
$ hadoop/bin/hadoop fs -cat /user/hadoop/security-test.txt
cat: org.apache.hadoop.security.AccessControlException: Permission
denied: user=garry, access=READ, inode="security-test.txt":hadoop:
supergroup:rw-------
```

(9) 把 hadoop 目录下的 security-test.txt 文件复制到 garry 用户根目录，并设置文件访问权限，使其只能被创建者访问。

```
$ Hadoop/bin/Hadoop fs -put security-test.txt security-test.txt
$ Hadoop/bin/Hadoop fs -chmod 700 security-test.txt
$ hadoop/bin/hadoop fs -ls
```

上述命令的输出如下图所示：



(10) 确认 garry 用户可以读取该文件内容。

```
$ hadoop/bin/hadoop fs -cat security-test.txt
```

你会在屏幕上看到下面这行字。

```
I can read this!
```

(11) 注销系统并返回到 hadoop 用户。

```
$ exit
```

(12) 尝试访问 garry 用户根目录下的 security-test.txt 文件。

```
$ hadoop fs -cat /user/garry/security-test.txt
```

你会在屏幕上看到下面这行字。

```
I can read this!
```

原理分析

我们首先使用hadoop用户在HDFS的根目录下创建一个测试文件。接下来，我们使用`hadoop fs`命令的`-chmod`选项修改文件权限，之前从未见过该选项。它与标准Unix的**chmod**工具非常相似，可以为文件创建者、某一用户组中所有成员以及所有用户指定文件的读/写/执行权限。

接着，我们再次使用hadoop用户登录到另一台主机并试图访问文件。毫不奇怪，文件访问成功。但是为什么呢？Hadoop凭什么来判断用户身份并决定是否允许其访问特定文件呢？

为了研究这个问题，我们随后在HDFS创建了另一个根目录（可以使用登录到主机上的任意账号），并通过`hadoop fs`命令的`-chown`选项为该目录赋予创建者权限。这个过程依然类似于标准Unix的`chown`工具。然后，我们切换到garry用户并试图访问存放在hadoop用户根目录下的文件。这次访问失败了，系统给出一个安全异常，这也符合我们的预期。接下来，我们把测试文件拷贝到garry用户的根目录下，并修改其权限，使其只能被创建者访问。

为了把事情弄乱，我们又切换回hadoop用户并试着访问garry用户根目录下的文件。令人奇怪的是，居然访问成功了。

1. 用户身份

第一个问题的答案是，Hadoop使用执行HDFS命令的用户的Unix ID作为HDFS上的用户身份。所以，用户`alice`执行命令创建的文件的所有者为`alice`,该用户只能读写其拥有相应访问权限的文件。

有安全意识的人会认识到，任何人只要在可以连接到集群的任意一台主机上创建一个与现有HDFS用户同名的用户，即可实现对Hadoop集群的访问。因此，在上个例子中，在任何一台可以访问NameNode的主机上创建的名为`hadoop`的用户都可以读取用户`hadoop`可访问的文件内容，这种情况非常糟糕。

- *超级用户*

上个例子中，我们看到hadoop用户访问了garry用户的文件。Hadoop把启动集群的用户ID视为超级用户，并赋予其多种特权，比如读、写、修改HDFS上任意文件的权限。有安全意识的人会认识到，这种风险甚至要比在Hadoop管理员无法控制的主机上随意创建名为`hadoop`的用户的风险更大。

2. 更细粒度的访问控制

在Hadoop发展的早期阶段，上述情况导致其安全性成为一个主要弱点。但是，Hadoop开发团队没有停滞不前，在完成大量工作之后，最新版的Hadoop支持更细粒度的、更强的安全模型。

为避免简单依赖于用户ID，开发者需要从其他地方获悉用户身份，他们选择了Kerberos系统。这需要建立并维护Kerberos系统，它们超出了本书范围，但如果这些安全机制对读者来讲很重要的话，请自行查阅Hadoop文档。请注意，用户可以综合使用基于Kerbros的身份认证机制和其他第三方认证系统，如微软活动目录（Microsoft Active Directory），因此其功能非常强大。

通过物理访问控制解决安全模型问题

如果读者认为Kerberos系统太过复杂，或者安全性是一个最好具备而非必须具备的功能，还可以通过一些其他办法降低安全风险。我喜欢的一个办法是，将整个集群部署在有着严格的访问控制策略的防火墙之后。尤其是，只允许集群中的一台主机访问NameNode和JobTracker服务，该主机被视为集群的头节点，所有用户都要连到该主机。



从集群外的主机访问Hadoop

使用命令行工具访问HDFS或运行MapReduce作业时，并不要求Hadoop处于运行状态。只要正确地安装了Hadoop，并在其配置文件中正确设置了NameNode或者JobTracker的位置，调用`Hadoop fs`和`Hadoop jar`命令时就会找到NameNode或者JobTracker节点。

这种模型的工作原理是，保证只有一台主机可以与Hadoop交互。因为该主机被集群管理员控制，正常用户无法创建或访问其他用户账号。

请记住，这种方法并不提供安全机制。它为一个脆弱的系统套上了坚硬的外壳，可以降低破坏Hadoop安全模型的可能性。

7.9 管理 NameNode

接下来，我们将讨论更多降低风险的有效途径。在第6章中，我曾以NameNode主机故障可能造成的严重后果为例吓唬过读者。如果那一节没有吓到你，回过头去再读一遍。概括地讲，如果集群中的NameNode无法提供服务，用户会失去存储在集群上的所有数据。这是因为NameNode负责向`fsimage`文件写入数据，该文件包含了文件系统的所有元数据并记录了哪个文件由哪些数据块组成。如果NameNode主机不再提供服务，会造成用户无法访问`fsimage`文件，其效果就像是丢失了所有HDFS数据。

为`fsimage`配置多个存储位置

用户可以配置多个`fsimage`文件的存储路径，这样NameNode就会同时向多个位置写入`fsimage`文件。这完全是一种冗余策略，多个存储设备只是用于存储相同数据的多个副本，而且其目的并不是为了提高性能。然而，通过这种策略，可以降低`fsimage`文件的多个副本同时丢失的可能性。

7.10　实践环节：为 fsimage 文件新增一个存储路径

为了满足数据恢复的需求，我们把 NameNode 配置成同时输出 fsimage 文件的多个副本。为了完成相应的设置，我们需要一个 NFS 导出目录。

(1) 确保集群已停止运行。

```
$ stopall.sh
```

(2) 在 Hadoop/conf/core-site.xml 文件中加入下列属性，把第二个路径改为 NFS 挂接位置，NameNode 的另一个副本数据可以写入该位置。

```
<property>
<name>dfs.name.dir</name>
<value>${hadoop.tmp.dir}/dfs/name,/share/backup/namenode</value>
</property>
```

(3) 删除新增路径下的已有内容。

```
$ rm -f /share/backup/namenode
```

(4) 启动集群。

```
$ start-all.sh
```

(5) 验证 fsimage 被写入指定的那两个位置，并为之前指定的这两个文件运行 md5sum 命令（根据你配置的路径修改下列代码）。

```
$ md5sum /var/hadoop/dfs/name/image/fsimage
a25432981b0ecd6b70da647e9b94304a    /var/hadoop/dfs/name/image/
fsimage
$ md5sum /share/backup/namenode/image/fsimage
a25432981b0ecd6b70da647e9b94304a    /share/backup/namenode/image/
fsimage
```

原理分析

首先，我们保证集群已经停止运行。尽管正在运行的集群不会重新读取修改过的核心配置文件的内容，但在配置之前停止集群运行是一个好习惯，可以防范Hadoop新加入重新读取核心配置文件这一功能。

接着，我们在集群配置文件中加入一个新属性：data.name.dir。该属性的值以逗号为分隔符，指明了fsimage文件的写入路径。请注意，data.name.dir反向引用了之前曾讨论过的hadoop.tmp.dir属性，其语法与Unix中的变量反向引用类似。使用该语法可以在其他属性值基础上扩展新的属性值，并可以继承父属性的更新。



别忘了加入需要的所有位置

data.name.dir属性的默认值是`${Hadoop.tmp.dir}/dfs/name`。在新增另一个属性值时，记得要明确地加入原来的默认值，如上所示。否则，该属性只会使用那个新值。

在启动集群之前，我们要确保存在新路径且该路径下没有数据。如果不存在该路径，NameNode的启动过程会失败，这是我们可以预料的。但是，如果曾在该目录下存储NameNode数据，启动过程同样会失败，因为两个路径下的NameNode数据不同，NameNode无法确定哪个才是正确的。

请注意！读者一定不想意外误删了某个目录中的数据，尤其是在试验使用多个NameNode数据存放位置，或者在节点间往复交换页面数据的时候。

在启动HDFS集群后，稍等片刻，然后使用MD5校验和来验证这些位置存放的`fsimage`文件完全相同。

`fsimage`副本的写入位置

我们推荐至少向两个位置写入`fsimage`文件，其中一个位置应当是远程文件系统，例如网络文件系统（NFS），就像上个例子那样。Hadoop系统只会定期更新`fsimage`文件的内容，因此不需要文件系统的性能有多高。

在前几节学习选配硬件的时候，我们暗示过读者要为NameNode选配特殊硬件。因为`fsimage`文件至关重要，有必要将其写入多个硬盘，也有必要为其购置高可靠性的硬盘，甚至可以将其写入RAID阵列。如果NameNode主机发生故障，最简单的办法就是使用写入远程文件系统的`fsimage`副本。但假如恰好远程文件系统也发生了故障，只好从宕机的NameNode上拔下硬盘并插入另一台主机恢复数据。

7

迁移到另一台NameNode主机

我们已确保`fsimage`写入多个位置，这是向另一台NameNode迁移的最重要的前提。现在我们来完成这个迁移过程。

切记不要在产品集群上执行这些操作。在产品集群上进行首次实验是被绝对禁止的，但即使不是第一次执行这些操作也不能说明这些操作是完全没有风险的。但一定要在其他集群上进行实验，从而知道在灾难袭来的时候应该怎么办。

在灾难来袭之前做好准备

读者一定不想在需要恢复产品集群时才来学习本节内容。提前做一些准备工作，不仅可以在灾难发生后恢复NameNode，而且可以使这个过程更轻松。

- 保证NameNode向多个位置写入`fsimage`文件。
- 决定在哪台主机上运行新NameNode。假如该主机目前被用作DataNode和TaskTracker，确保其硬件配置满足NameNode的需求，并且不会因缺少一台工作主机而造成集群性能大幅下降。
- 复制`core-site.xml`和`hdfs-site.xml`，最好把它们放到网络文件系统（NFS）上，并修改这些文件内容，使其指向新NameNode主机。在对现有配置文件进行修改的时候，也要及时在这些配置文件副本中进行相同修改。
- 把位于NameNode主机上的`slaves`文件拷贝到新NameNode主机或者NFS的共享文件夹。同样，要保证对它进行同步更新。
- 明白如何处理新主机出现的故障。修复或替换原来的发生故障的主机需要多长时间？在此期间，哪台主机将作为下一台承载NameNode和SecondaryNameNode的主机？

准备好了吗？让我们开始吧！

7.11　实践环节：迁移到新的 NameNode 主机

下面的步骤中，我们把新配置文件放在挂接到 `share/backup` 的一个 NFS 共享文件夹中，并根据新文件的存储位置改变配置文件中的相应内容。另外，我们在配置文件中查找新 NameNode 主机的 IP 地址，该 IP 地址专用于 NameNode 主机。

(1) 登录到目前的 NameNode 主机，停止集群运转。

```
$ stop-all.sh
```

(2) 关闭运行着 NameNode 进程的主机。

```
$ sudo poweroff
```

(3) 登录到承载新 NameNode 进程的主机，并确认新配置文件中的 NameNode 位置是正确的。

```
$ grep 110 /share/backup/*.xml
```

(4) 把`slaves`文件拷贝到新NameNode主机上。

```
$ cp /share/backup/slaves Hadoop/conf
```

(5) 把更新过的配置文件拷贝到新 NameNode 主机上。

```
$ cp /share/backup/*site.xml Hadoop/conf
```

(6) 删去本地文件中的旧 NameNode 的数据。

```
$ rm -f /var/Hadoop/dfs/name/*
```

(7) 把更新过的配置文件拷贝到集群中的每个节点。

```
$ slaves.sh cp /share/backup/*site.xml Hadoop/conf
```

(8) 保证每个节点上有一个指向新 NameNode 节点的配置文件。

```
$ slaves.sh grep 110 hadoop/conf/*site.xml
```

(9) 启动集群。

```
$ start-all.sh
```

(10) 通过命令行检查 HDFS 的运行状态。

```
$ Hadoop fs ls /
```

(11) 验证是否可通过网页用户接口访问 HDFS。

原理分析

首先，我们关掉了整个集群。这个操作不具有代表性，因为由故障导致NameNode停止运行的大多数情况下，NameNode停止运行的方式都不太友好。但本章后面几节会讲到文件系统错误引发的问题。

接着，我们关掉了旧NameNode主机。严格意义上来讲，这不是必要的，但它保证了其他节点无法访问该主机。它给你一种错觉，认为向新NameNode主机的迁移过程很顺利。

在把配置文件拷贝到新主机之前，我们快速浏览一下`core-site.xm`和`hdfs-site.xml`，确认`core-site.xml`中的`fs.default.dir`属性值是正确的。

之后，我们在新主机上进行准备工作。首先把`slaves`配置文件以及集群配置文件拷贝至新主机，然后把本地路径下的旧NameNode数据删除。参阅前面的步骤，本步骤要特别小心。

接下来，我们使用`slaves.sh`脚本把新配置文件拷贝到集群中每个节点上。我们知道，只有新NameNode主机的IP中包含字符串“110”，因此我们在文件中查找该字符串以确保所有配置都是最新的（很明显，你需要在你的系统中使用一个不同的字符串）。

至此，所有操作都已完成。我们启动集群并通过命令行工具和网页用户接口访问集群，以确认集群按照我们预期的方式运转。

1. 不要高兴得太早

请记住，即使成功地迁移到了新NameNode主机，事情还没有完成。你需要提前决定如何处理SecondaryNameNode，以及假如NameNode再次出现故障，下一步要把NameNode迁移到哪台主机上。为了做好这些准备，你需要再次浏览刚刚提到的“准备工作”清单并据此开展准备工作。



> 不要忘了考虑发生关联故障的可能性。及时调查NameNode主机发生故障的原因，否则它将再次引发更大的问题。

2. 如何完成JobTracker迁移过程

我们没有提如何迁移JobTracker，因为第6章曾讲过，这个过程相对简单一些。如果NameNode和JobTracker运行在同一台主机上，需要改进上述方法，即更新`mapred-site.xml`副本的内容，使其中的`mapred.job.tracker`属性指向新JobTracker主机的位置。

一展身手：把NameNode迁移到一台新主机

实践一下，把NameNode和JobTracker从一台主机迁移到另一台主机上。

7.12　管理 HDFS

在第6章中曾讲到过，杀死节点并重启时，Hadoop会自动解决很多可用性问题。如果在传统文件系统上，这些问题会耗费集群管理员很多精力。虽然Hadoop自动实现了这些功能，但我们仍需了解其工作原理。

7.12.1　数据写入位置

就像可以通过`dfs.name.dir`属性为NameNode的`fsimage`文件指定多个存储位置一样，我们曾经提到过，有一个名为`dfs.data.dir`的类似属性，它允许在HDFS使用多个数据存储位置。本节我们将学习相关内容。

这是一种有效方案，其工作原理与NameNode的原理有较大区别。如果在`dfs.data.dir`属性值中指定多个路径，Hadoop会把它们当做可并行使用的独立位置。如果用户拥有多个指向文件系统不同位置的物理硬盘或其他存储设备时，这个属性非常有用。Hadoop会智能调度这些设备，不仅可使存储总量最大化，同时将读写操作均匀分配到这些设备上，以获得最大吞吐量。在7.4.3节曾提到，这种方法可以达到存储量和吞吐量最大化，但却以健壮性为代价，单个硬盘故障就会导致主机失效。

7.12.2 使用平衡器

Hadoop尽量优化HDFS上的数据块存放策略，以达到最佳性能和最大冗余。但是，在某些情况下，集群出现了失衡现象，各节点存储的数据量有着较大差异。最典型的情况就是集群中出现新增节点的时候。默认情况下，Hadoop认为新增节点和其他节点地位相同，这就意味着，在相当长一段时间内，新增节点的空间使用率会维持在较低水平。失效或出现其他问题的节点上存储的数据块也明显少于其他节点。

Hadoop提供了一个称为平衡器的工具来解决这个问题。我们可以通过分别运行`start-balancer.sh`和`stop-balancer.sh`脚本启用或停用平衡器。

运行平衡器的时机

Hadoop系统中缺乏一种提醒用户文件系统已失衡的自动报警机制。相反，用户需要自己留意`hadoop fsck`和`hadoop fsadmin`反馈的数据，观察节点间是否存在失衡现象。

实际上，用户不必太过担心这个问题，因为Hadoop有着完善的数据块放置策略，只有当新增硬件或修复故障节点时，用户才需要运行平衡器消除主要的失衡现象。但是，为了最大程度地维护集群正常运转，人们通常会按照预定计划定期执行平衡器，例如，每晚执行一次，这样会把数据块的平衡系数维持在用户设定的水平上。

7.13 MapReduce 管理

从上一章可以看出，一般来讲，MapReduce框架比HDFS的容错能力更强。因为JobTracker和TaskTracker不需要维护永久数据，因此，对MapReduce的管理更多的是指对正在运行的作业和任务的管理，而不是维护框架本身。

7.13.1 通过命令行管理作业

实现作业管理的主要工具是`Hadoop job`命令行工具。像往常一样，输入下列命令获取其使用说明。

```
$ hadoop job --help
```

该命令的大部分选项都无需解释。它支持启动、终止、列出运行中的作业，以及修改运行中的作业。此外，还可以通过该命令查询作业的历史状态。下节内容不会逐一解释这些选项，而是通过在一个例子中练习多个子命令来研究其用法。

一展身手：通过命令行管理作业

使用MapReduce的网页用户接口也可以访问部分上述功能。研究一下MapReduce的网页用户接口，弄清楚用户通过该接口可以实现和无法实现哪些功能。

7.13.2　作业优先级和作业调度

截至目前，我们通常只在集群中运行一个作业并等至其运行结束。这就掩盖了一个重要事实：默认情况下，Hadoop将提交的后续作业放入一个**FIFO**（First In, First Out，先入先出的队列）。在一个作业结束后，Hadoop会启动队列中的下一个作业。除非我们选用了后面几节会讲到的另一个调度算法，采用先入先出调度算法的集群专用于处理正在运行的单个作业。

对于很少有作业排队等候运行的小集群来说，这种方式完全没问题。但是，如果队列中经常有一些作业在等待执行机会，便会产生问题。特别是，FIFO调度模型没有考虑作业优先级或者作业所需的资源。一个运行时间较长但优先级较低的作业会先于运行时间较短而优先级较高的作业运行，仅仅因为前者的提交时间早于后者。

为了解决这个问题，Hadoop定义了5种不同程度的作业优先级，它们分别是：VERY_HIGH、HIGH、 NORMAL、LOW和VERY_LOW。作业的默认优先级是NORMAL，但可以通过hadoop job -set-priority命令修改它。

7.14　实践环节：修改作业优先级并结束作业运行

接下来，我们会动态修改作业优先级，并观察强制结束正在运行的作业后，哪个作业会得到执行机会。

(1) 在集群上启动一个需要运行很长一段时间的作业。

```
$ hadoop jar hadoop-examples-1.0.4.jar pi 100 1000
```

(2) 打开另一个窗口并提交第二个作业。

```
$ hadoop jar hadoop-examples-1.0.4.jar wordcount test.txt out1
```

(3) 打开另一个窗口并提交第三个作业。

```
$ hadoop jar hadoop-examples-1.0.4.jar wordcount test.txt out2
```

(4) 列出正在运行的作业。

```
$ Hadoop job -list
```

你会在屏幕上看到下列输出。

```
3 jobs currently running
JobId    State    StartTime    UserName    Priority    SchedulingInfo
job_201201111540_0005  1  1326325810671  hadoop   NORMAL  NA
job_201201111540_0006  1  1326325938781  hadoop   NORMAL  NA
job_201201111540_0007  1  1326325961700  hadoop   NORMAL  NA
```

(5) 检查正在运行的作业的状态。

```
$ Hadoop job -status job_201201111540_0005
```

你会在屏幕上看到如下输出。

```
Job: job_201201111540_0005
file: hdfs://head:9000/var/hadoop/mapred/system/
job_201201111540_0005/job.xml
tracking URL: http://head:50030/jobdetails.
jsp?jobid=job_201201111540_000
map() completion: 1.0
reduce() completion: 0.32666665
Counters: 18
```

(6) 把最后提交的作业的优先级提高为 VERY_HIGH。

```
$ Hadoop job -set-priority job_201201111540_0007 VERY_HIGH
```

(7) 强制结束正在运行的作业。

```
$ Hadoop job -kill job_201201111540_0005
```

(8) 观察其余作业，看哪个作业开始运行。

7

原理分析

我们在集群上启动了一个作业，并连续提交另两个作业，使它们处于排队等候状态。然后通过hadoop job -list命令确认队列中的作业顺序和我们的预期一致。hadoop job-list all命令会列出所有已完成作业和目前正在运行的作业，而hadoop job -history会允许我们更详细地检查作业及其任务的信息。为了确认已提交作业处于运行状态，我们使用hadoop job -status获取作业中map和reduce任务的完成进度，以及作业计数器的状态。

接着，我们使用hadoop job -set-priority提高队列中最后一个作业的优先级。

使用hadoop job -kill命令强制结束正在运行的作业后，我们确认下一个要运行的是刚提升优先级的作业，即使留在队列中的作业的提交时间要比它早。

另一种调度器

手动修改FIFO队列中作业的优先级的办法确实有效，但它需要主动监测并管理作业队列。仔细想想这个问题,我们发现出现这种局面的原因在于Hadoop将集群资源全部分配给了正在执行的单个作业。

Hadoop提供了另外两种作业调度器，它们采用了不同的方法，可以在多个同时运行的作业之间共享集群。Hadoop还提供了一种添加额外调度器的插件机制。请注意，资源共享是一个在概念上简单，而在实现上非常复杂的问题，很多学术研究集中在这个领域。其目标不光是在特定时间点提高资源分配率，并且要在一段时间内优先运行具有较高优先级的作业。

1. 计算能力调度器

计算能力调度器采用多个作业队列，每个队列都会获得一部分集群资源。用户提交的作业会分别出现在各个队列中。例如，用户可以为运行时间较长的作业维护一个较大的队列，并为它分配90%的集群资源，同时为优先级较高的作业维护一个较小的队列，并为该队列分配10%的集群资源。如果每个队列中都有一些待执行的已提交作业，集群资源就按此比例进行分配。

但是，如果一个队列中的所有作业都已执行完毕，而另一个队列中还有待执行的作业，计算能力调度器会暂时将空队列的资源分配给忙队列。一旦作业被提交至空队列，该队列在完成正在运行的作业之后会再次获得其原有容量。该方法在提高资源分配率和防止资源长期闲置这两个方面取得了合理的平衡。

计算能力调度器为各队列实现了优先级功能，尽管默认情况下，这一功能是禁用的。即使高优先级作业的提交时间晚于低优先级作业，系统会在出现可用计算资源时优先执行高优先级作业。

2. 公平调度器

公平调度器把整个集群分割成若干个资源池，系统将用户提交的作业分配到各个资源池中，并且通常用户和资源池之间存在某种关联。尽管默认情况下，各个资源池获得份额相等的集群资源，但我们可以修改资源分配比例。

在各资源池中，默认模式是所有提交到该池的作业都共享其资源。因此，假设集群被分成Alice和Bob两个资源池，其中每个池中都有3个作业，那么集群会并行执行这6个作业。用户可以限制资源池中并行作业的总数，因为同时运行太多作业会产生大量临时数据，总的来说会降低作业的运行效率。

与计算能力调度器一样，假如某个资源池中的所有作业都已执行完毕，公平调度器就会将该池的集群资源分配给其他资源池，并在该池重新收到作业时收回自己的资源。它也支持作业优先级，它会优先安排执行高优先级作业，而不管它的提交时间是否早于低优先级作业。

3. 启用替代调度器

Hadoop安装路径下的`contrib`路径中包括两个名为`capacityScheduler`和`fairScheduler`的子目录，其中包含一些以JAR文件形式存在的替代调度器。为了启用替代调度器，要么将其JAR文件添加到`hadoop/lib`目录下，要么明确地在classpath中加入文件位置。请注意，用户需要分别为各个调度器配置其属性。查阅文档以获取更详细的信息。

4. 何时使用替代调度器

替代调度器非常有用，但在小规模集群、无需并发运行多个作业的集群或者无需保证优先执行高优先级作业的情况下，并不需要使用替代调度器。每个调度器都有多个配置参数，用户需要调整这些参数以达到集群资源的最佳利用率。但对有着多个用户和多种作业优先级的大规模集群，替代调度器必不可少。

7.15 扩展集群规模

目前，用户已经搭建了一个Hadoop集群，并用它来处理手头的数据。但是，随着数据越来越多，需要更多的集群资源来处理这些数据。我们曾经反复说过，Hadoop是一个易扩展系统。接下来，我们将扩展集群，提升其计算能力。

7.15.1 提升本地Hadoop集群的计算能力

我希望读者不要担心如何为正在运行的集群新增节点这个问题。在第6章中，我们不断地杀死节点并重启。与之相比，新增节点并没有什么特别之处，用户只需执行下列步骤即可。

(1) 在主机上安装Hadoop。

(2) 按照第2章中讲到的步骤设置环境变量。

(3) 把配置文件拷贝到安装路径下的`conf`目录中。

(4) 把主机的DNS名或IP地址加到`slaves`文件中，该文件位于用户执行`slaves.sh`、`start-all.sh`或`stop-all.sh`的节点上。

这就是所需的全部步骤。

一展身手：加入新节点并运行平衡器

尝试在集群中新增一个节点，事后检查HDFS状态。如果HDFS处于失衡状态，运行平衡器修复失衡现象。为了让这种效果更明显，可以在新增节点之前，先在HDFS上存储大量数据。

7.15.2　提升EMR作业流的计算能力

如果用户使用的是弹性MapReduce，对非持久性集群而言，并不存在扩展集群规模的概念。因为在设置作业流时，用户就指定了所需的主机数和主机类型，用户只需保证集群规模与待执行的作业相匹配即可。

扩展正在运行的作业流

但是，有时用户希望尽快执行完一个所需时间较长的作业。在这种情况下，用户需要为处于运行状态的作业流加入更多节点。回想一下，EMR有3种不同类别的节点：运行NameNode和JobTracker的主节点，运行HDFS的核心节点，以及运行MapReduce作业的任务节点。在此情况下，用户可以为作业流新增任务节点到达更快执行MapReduce作业的目的。

另一种情况是，用户定义的作业流包括一系列而不只是一个MapReduce作业。EMR允许分别修改各个作业流的配置。这样做的好处是，为每个作业提供了量身定制的硬件配置，能够更好地控制性能和成本，在二者之间取得平衡。

标准的EMR数据处理模型是，作业流从S3读取源数据，在临时的EMR Hadoop集群上处理这些数据，然后再把结果写回到S3。但是，如果用户的数据集规模很大而且需要频繁处理，反复拷贝数据是一项非常耗时的工作。在这种情况下，可以使用另一种模型，那就是在作业流中使用一个持久性的Hadoop集群，为其配备足够多的核心节点从而在HDFS存储所需数据。在处理数据时，像刚才说的那样通过为作业流分配更多任务节点达到提升计算能力的目的。



目前，AWS控制台不支持为正在运行的作业流调整集群规模，用户需要通过API或命令行工具实现这一功能。

7.16　小结

本章讲述了如何搭建、维护和扩展Hadoop集群。特别是，我们知道了哪个文件设置了Hadoop配置属性的默认值，以及如何通过编写代码实现作业级的属性设置。我们也学习了如何为集群选配硬件，购买硬件之前评估工作负载的重要意义，以及Hadoop如何获得主机所处机柜的物理位置，从而优化数据块放置策略。

接着，我们了解了默认的Hadoop安全模型的工作原理，它的弱点以及应对之策。我们还学习了如何降低NameNode的故障风险（见第6章），如何在灾难袭来之时将NameNode迁移到一台新主机。我们也学习了数据块副本放置策略，集群失衡的原因以及如何应对集群的失衡状态。

我们还研究了Hadoop提供的MapReduce作业调度模型，学习了作业优先级如何改变作业执行顺序，计算能力调度器和公平调度器如何以更复杂的方式为多个并发作业分配集群资源，以及如何扩展集群以提高其计算能力。

本书对Hadoop核心内容的研究到此为止 。在其余章节中，我们会接触一些建立在Hadoop基础上的其他系统和工具，它们可以帮助用户更精细地理解数据，并与其他系统协同工作。在下一章中，我们将学习使用Hive，用类似关系数据库的概念处理HDFS上的数据。

第 8 章

Hive：数据的关系视图

MapReduce是一个功能强大的数据处理范式，能从复杂的数据处理过程中凝练出宝贵的结论。但是，它把数据处理分析过程拆分成一系列map和reduce阶段，需要用户接受这种理念，进行相应的训练并有一定的经验。借助一些建立在Hadoop基础上的产品，用户能从更高或更熟悉的角度理解存储在HDFS上的数据。本章将介绍其中最流行的一款工具，它就是Hive。

本章包括以下内容：

- 什么是Hive以及使用Hive的原因；
- 如何安装并配置Hive；
- 使用Hive对UFO数据集执行类SQL分析；
- Hive与关系数据库的共同特点，如联结和视图；
- 怎样有效地将Hive应用于特大数据集；
- Hive如何在查询语句中融入用户自定义函数；
- Hive与另一款常用工具Pig的互补关系。

8.1 Hive 概述

Hive是建立在Hadoop基础上的数据仓库，它使用MapReduce对存储于HDFS上数据进行分析。它专门定义了一种类SQL（Structured Query Language）查询语言，我们称之为**HiveQL**。

8.1.1 为什么使用Hive

在第4章中，我们介绍了Hadoop Streaming。它的一个最大优势在于缩短了MapReduce作业的开发周期。Hive可以进一步缩短开发周期。它没有提供更快地开发map和reduce任务的方法，而

是定义了一种类SQL查询语言。Hive使用HiveQL语句表述查询操作，并立刻将其自动转化成一个或多个MapReduce作业，然后执行这些MapReduce程序并将结果反馈给用户。Hapoop Streaming缩短了“编码/编译/提交”的开发周期，而Hive则完全摒弃了这一过程，只需构造HiveQL语句即可。

Hadoop的这个接口不仅加速了从分析数据到生成结果的过程，而且明显拓宽了Hadoop和MapReduce的使用人群。用户要想使用Hadoop，需要首先掌握软件开发技术。而现在，任何熟悉SQL的用户都可以使用Hive。

因为Hive同时具备上述特性，用户经常用它进行业务和数据分析，并对存储在HDFS上的数据执行特殊查询。直接使用MapReduce需要在执行作业前编写map和reduce任务，这就意味着，从最初产生某种查询想法到实现该想法的过程会产生无法避免的延迟。而使用Hive，用户只需编写精炼的HiveQL查询语句就可直接进行数据分析工作，不再需要软件开发人员一直参与数据分析过程。当然，Hive技术在操作层面也存在一些实际的限制（不管该技术功能多么强大，糟糕的查询语句的执行效率很低）。但其适用范围广，这一点还是很吸引人的。

8.1.2　感谢Facebook

我们曾因为Google、Yahoo!和Doug Cutting对Hadoop的杰出贡献深表感谢，现在，我们必须衷心感谢Facebook。

最初，Hive是由Facebook的数据组开发维护的，在Facebook内部使用之后，它被移交给Apache软件基金会，此后，人们可以像使用开源软件一样免费使用Hive。它的主页地址是http://hive.apache.org。

8.2　设置 Hive

本节我们将介绍如何下载、安装并配置Hive。

8.2.1　准备工作

与Hadoop不同，Hive系统中不存在主节点、从节点或工作节点。Hive以客户端应用程序的形式运行，负责处理HiveQL查询，将查询语句转化为MapReduce作业，并将作业提交到一个Hadoop集群。

虽然Hive提供了一种适用于小作业和开发使用的方式，但通常情况下，Hive需要一个现有的正在运行的Hadoop集群来配合运行MapReduce作业。

正如其他Hadoop客户端不需要在实际的集群节点上执行，Hive可以在符合下列条件的任何主机上执行：

- 安装了Hadoop的主机（即使主机上没有正在运行的进程）；
- 把HAPOOP_HOME环境变量的值设置为Hadoop安装目录的主机；
- 系统路径或用户路径中出现${HADOOP_HOME}/bin目录的主机。

8.2.2 下载Hive

读者可以从http://hive.apache.org/releases.html下载到Hive的最新稳定版本。

Hive入门指南（其网址为http://cwiki.apache.org/confluence/display/Hive/GettingStarted）会从兼容的角度为用户推荐合适的版本，但一般情况下，可以预见，Hive、Hadoop和Java的最新稳定版应当能够协同工作。

8.3 实践环节：安装 Hive

接下来，我们将安装并配置 Hive，以便日后使用。

(1) 下载 Hive 的最新稳定版本，并将其放到安装目录下。

```
$ mv hive-0.8.1.tar.gz /usr/local
```

(2) 解压安装包。

```
$ tar -xzf hive-0.8.1.tar.gz
```

(3) 把安装路径设为变量 HIVE_HOME 的值。

```
$ export HIVE_HOME=/usr/local/hive
```

(4) 将 HIVE 的主目录添加到 PATH 变量中：

```
$ export PATH=${HIVE_HOME}/bin:${PATH}
```

(5) 在 HDFS 上创建 Hive 所需路径/tmp 和/user/hive/warehouse。

```
$ hadoop fs -mkdir /tmp
$ hadoop fs -mkdir /user/hive/warehouse
```

(6) 修改上述路径的访问权限，使用户组具有写入权限。

```
$ hadoop fs -chmod g+w /tmp
$ hadoop fs -chmod g+w /user/hive/warehouse
```

(7) 试着启动 Hive。

```
$ hive
```

该命令的执行结果如下所示。

```
Logging initialized using configuration in jar:file:/opt/hive-
0.8.1/lib/hive-common-0.8.1.jar!/hive-log4j.properties


Hive history file=/tmp/hadoop/hive_job_log_
hadoop_201203031500_480385673.txt
hive>
```

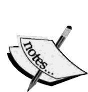
(8) 退出 Hive 的交互式 shell。

```
$ hive> quit;
```

原理分析

下载到Hive的最新稳定版安装包之后，我们将其拷贝至安装位置并解压文件。这一步会自动创建一个名为hive-<version>的目录。

与之前定义HADOOP_HOME并将安装目录下的bin路径添加到path变量类似，我们定义了HIVE_HOME并将Hive的bin路径添加至path变量。



请记住，为了避免用户每次登录都要设置这些变量，可以把它们添加到用户登录的shell脚本或一个单独的配置脚本中，这样用户使用Hive时只需调用这些脚本即可。

接着，我们在HDFS上创建了两个Hive要用到的目录，并修改它们的属性，使用户组具有对该目录的写入权限。默认情况下，Hive会把执行查询语句产生的临时数据写入/tmp目录，除此之外，输出数据也会被写入该目录。/user/hive/warehouse目录则用于存储写入Hive表中的数据。

完成这些设置之后，我们运行hive命令。如果安装成功的话，该命令的输出与上述输出类似。不带任何参数运行hive命令，会进入一个交互shell。hive>提示符类似于关系数据库中的交互工具sql>或mysql>。

接着，我们输入quit;命令退出交互shell。一定要注意末尾的分号;。如前所述，HiveQL与SQL非常相似，并遵循SQL中关于"所有命令必须以分号结束"的约定。没有输入分号而按下回车键，意味着用户将在下一行继续输入命令。

8.4 使用 Hive

成功安装Hive之后，我们将用它来分析第4章介绍过的UFO数据集。

向Hive导入新数据的过程通常分为以下3个阶段。

(1) 定义表的各个字段，该表将用于导入数据。
(2) 把数据导入已创建的表中。
(3) 针对上表执行HiveQL查询。

上述过程与关系数据库中的操作非常相似。Hive支持数据的结构化查询视图，在执行任何查询之前，我们必须首先为表的每列定义规范并把数据导入表中。



> 我们假设读者较为熟悉SQL语言，所以本章内容重在介绍怎样使用Hive处理数据，而不会详细解释SQL的某个概念。不太熟悉SQL语言的读者可以在手边准备一本SQL参考手册，虽然我们保证读者能够理解每条SQL语句的功能，但有些语句的细节可能需要读者深入研究。

8.5　实践环节：创建 UFO 数据表

接下来，通过下列步骤新建一个表，并把 UFO 数据导入该表。

(1) 启动 Hive 的交互式 shell。

```
$ hive
```

(2) 为 UFO 数据集创建表。为了增强可读性，我们将相关语句分成多行显示。

```
hive> CREATE TABLE ufodata(sighted STRING, reported STRING,
sighting_location STRING,    > shape STRING, duration STRING,
description STRING COMMENT 'Free text description')
COMMENT 'The UFO data set.' ;
```

输入上述语句之后，Hive 的输出结果如下所示。

```
OK
Time taken: 0.238 seconds
```

(3) 列出所有现有表。

```
hive> show tables;
```

该命令的执行结果如下所示。

```
OK
ufodata
Time taken: 0.156 seconds
```

(4) 显示与正则表达式“.*data”匹配的表。

```
hive> show tables '.*data';
```

该命令的执行结果如下所示。

```
OK
ufodata
Time taken: 0.065 seconds
```

(5) 验证表中各字段的定义。

```
hive> describe ufodata;
```

该命令的执行结果如下所示。

```
OK

sighted   string
reported  string
sighting_location    string
shape    string
duration  string
description    string    Free text description
Time taken: 0.086 seconds
```

(6) 更详细地显示对表的描述。

```
hive> describe extended ufodata;
```

该命令的执行结果如下所示。

```
OK

sighted   string
reported  string
...
Detailed Table Information  Table(tableName:ufodata,
dbName:default, owner:hadoop, createTime:1330818664,
lastAccessTime:0, retention:0,
...

...location:hdfs://head:9000/user/hive/warehouse/
ufodata, inputFormat:org.apache.hadoop.mapred.
TextInputFormat,outputFormat:org.apache.hadoop.hive.ql.io.
HiveIgnoreKeyTextOutputFormat, compressed:false, numBuckets:-1,
```

8

原理分析

启动Hive交互程序后，我们用CREATE TABLE命令来定义UFO数据表的结构。和标准SQL一样，HiveQL要求为表中的每列指定列名和数据类型。HiveQL还可以为每列和整个表加入可选注

释，上例中，我们为“description”列和全表加入了注释。

针对UFO数据，我们定义该表中所有字段的数据类型为`STRING`。和SQL一样，HiveQL也支持多种数据类型。

- ❑ **布尔类型**：`BOOLEAN`
- ❑ **整数类型**：`TINYINT`、`INT`、`BIGINT`
- ❑ **浮点类型**：`FLOAT`、`DOUBLE`
- ❑ **文本类型**：`STRING`

创建表之后，使用`SHOW TABLES`语句来确认表已经创建成功。该命令会列出系统中所有的表，在我们这个例子中，系统中只有一个UFO表。

接着，我们使用`SHOW TABLES`语句的变体，该语句附带了一个可选的JAVA正则表达式，用于列出与正则表达式匹配的所有表。本例中，输出结果与上个命令完全相同，但当系统中存在许多表的时候，尤其是不知道表的准确名称的时候，这种变体非常有用。



> 我们已成功创建了表，但还没有验证该表的结构是否合理。下一步，我们将使用`DESCRIBE TABLE`命令显示特定表的详细信息。我们看到，所有字段都和预期相同（尽管该命令不能返回表的注释信息）。接下来，我们使用`DESCRIBE TABLE EXTENDED`命令获取该表的更多信息。

上例中，我们略去了最终输出数据的大部分内容，仅展示其中有意思的一小部分。请注意，输入格式被指定为`TextInputFormat`。默认情况下，Hive假设插入表中的所有HDFS文件都以文本文件的形式存在。

同时，我们还观察到，表数据存储在之前创建的HDFS目录`/user/hive/warehouse`下。



> **关于字符大小写的提示**
>
> 就像SQL一样，HiveQL对关键词、列名、表名中的字符不区分大小写。按照惯例，SQL语句中的关键词使用大写字母，我们通常会在编写脚本文件中的HiveQL语句时遵照这个惯例，稍后会在具体示例中看到。但是，在输入交互式命令时，我们通常会选用最简单的方式——使用小写字符。

8.6　实践环节：在表中插入数据

上一节我们完成了表的创建工作，接下来，我们将把 UFO 数据导入 `ufodata` 表。

(1) 将 UFO 数据文件拷贝至 HDFS。

```
$ hadoop fs -put ufo.tsv /tmp/ufo.tsv
```

(2) 确认文件已成功复制到 HDFS。

```
$ hadoop fs -ls /tmp
```

上述命令的结果如下所示。

```
 Found 2 items
drwxrwxr-x    - hadoop supergroup    0 ... 14:52 /tmp/hive-
hadoop
-rw-r--r--    3 hadoop supergroup    75342464 2012-03-03 16:01 /tmp/
ufo.tsv
```

(3) 进入 Hive 交互式 shell。

```
$ hive
```

(4) 把步骤 1 复制到 HDFS 的文件数据插入 `ufodata` 表中。

```
hive> LOAD DATA INPATH '/tmp/ufo.tsv' OVERWRITE INTO TABLE
ufodata;
```

上述命令的结果如下所示。

```
Loading data to table default.ufodata
Deleted hdfs://head:9000/user/hive/warehouse/ufodata

OK
Time taken: 5.494 seconds
```

(5) 退出 Hive shell。

```
hive> quit;
```

(6) 检查 HDFS 上存放 UFO 数据副本的目录。

```
$ hadoop fs -ls /tmp
```

上述命令的结果如下所示。

```
Found 1 items
drwxrwxr-x    - hadoop supergroup    0 ... 16:10 /tmp/hive-
hadoop
```

原理分析

首先，我们把第4章中用到的以tab键分隔的UFO数据文件拷贝至HDFS。确认HDFS上存有文

件数据后，我们启动Hive交互shell并用LOAD DATA命令将文件数据载入ufodata表。

因为我们使用的文件已经放到了HDFS上，所以单独使用INPATH关键词来指定源文件的位置。我们还可以通过LOCAL INPATH指定位于本地文件系统上的源文件，将它直接导入Hive表中。这样就不必明确地将本地文件系统上的源文件拷贝到HDFS。

在把UFO数据导入ufodata表的Hive语句中，我们指定了OVERWRITE关键词，它会在导入新数据前删除表中原有数据。从该命令的执行结果可以看出，使用OVERWRITE会把存放表数据的目录清空，因此要谨慎使用该语句。

请注意，Hive系统用了5秒多时间来执行该命令，明显多于把UFO数据文件拷贝至HDFS的时间。



虽然我们在本例中明确使用了一个源文件，但我们通过指定一个目录作为INPATH的参数，使用一条LOAD命令导入多个文件。在这种情况下，目录中的所有文件都会导入到表中。

退出Hive shell之后，我们再次查看刚刚存放ufo数据文件副本的目录，结果发现该目录已被删除。如果传给LOAD语句的是HDFS上的数据路径，那么LOAD语句不光会将数据复制到/user/hive/datawarehouse，同时也会删掉其原始目录。如果读者想分析HDFS上被其他程序使用的数据，要么备份一个副本，要么使用后面将讲到的EXTERNAL方案。

8.6.1 验证数据

在将数据导入Hive表之后，一种好的做法是，立刻进行一些快速验证查询，以确定所建的表及表中数据和预期一致。有时侯，通过验证查询发现，表的初始定义就是错误的。

8.7 实践环节：验证表

进行初步验证的最简单方法就是，执行一些统计查询以验证导入数据是否正确。这与第 4 章使用 Hadoop Streaming 执行的任务类似。

(1) 在 hive 命令行工具中输入以下 HiveQL 语句，它会统计表中的记录总数，记得不要使用 Hive shell。

```
$ hive -e "select count(*) from ufodata;"
```

该命令的执行结果如下所示。

```
Total MapReduce jobs = 1
```

```
Launching Job 1 out of 1
...
Hadoop job information for Stage-1: number of mappers: 1; number
of reducers: 1
2012-03-03 16:15:15,510 Stage-1 map = 0%,    reduce = 0%
2012-03-03 16:15:21,552 Stage-1 map = 100%,    reduce = 0%
2012-03-03 16:15:30,622 Stage-1 map = 100%,    reduce = 100%
Ended Job = job_201202281524_0006
MapReduce Jobs Launched:
Job 0: Map: 1    Reduce: 1    HDFS Read: 75416209 HDFS Write: 6
SUCESS
Total MapReduce CPU Time Spent: 0 msec
OK
61393
Time taken: 28.218 seconds
```

(2) 作为示例，从 sighted 列选取 5 个值。

```
$ hive -e "select sighted from ufodata limit 5;"
```

该命令的执行结果如下所示。

```
Total MapReduce jobs = 1
Launching Job 1 out of 1
...
OK
19951009    19951009    Iowa City, IA    Man repts. witnessing
&quot;flash, followed by a classic UFO, w/ a tailfin at
back.&quot; Red color on top half of tailfin. Became triangular.
19951010    19951011    Milwaukee, WI    2 min.    Man      on Hwy 43 SW
of Milwaukee sees large, bright blue light streak by his car,
descend, turn, cross road ahead, strobe. Bizarre!
19950101    19950103    Shelton, WA    Telephoned Report:CA
woman visiting daughter witness discs and triangular ships over
Squaxin Island in Puget Sound. Dramatic.    Written report, with
illustrations, submitted to NUFORC.
19950510    19950510    Columbia, MO    2 min.    Man repts. son&apos;s
bizarre sighting of small humanoid creature in back yard.    Reptd.
in Acteon Journal, St. Louis UFO newsletter.
19950611 19950614   Seattle, WA      Anonymous caller repts.
sighting 4 ufo&apos;s in NNE sky, 45 deg. above horizon. (No
other facts reptd.  No return tel. #.)
Time taken: 11.693 seconds
```

原理分析

本例中，我们没有使用Hive交互式shell，而是直接在`hive -e`命令中使用HiveQL语句。交互式shell一般用于处理一系列的Hive操作。对一些简单的语句，直接把HiveQL查询语句传入命令

行工具更方便简洁。也就是说，在脚本中也可以调用Hive。



> 当使用hive-e时，没必要再用分号作为HiveQL语句的结束符，但有时候很难改掉这个习惯。如果你想在同一语句中执行多个命令，就必须用分号分开这些命令。

第一次查询的结果是61393，与我们之前直接用MapReduce分析UFO数据集时得到的结果是一样的。这表明整个数据集确实已被导入表中。

接着，我们执行了第二次查询，即从表的第一列选取5个值，我们希望它返回UFO出现的5个具体日期。但结果却是5条包括UFO出现日期在内的完整记录。

出现这种问题的原因在于，我们依靠Hive把数据文件以文本文件的形式导入表中，却没有考虑各列之间的分隔符。我们的数据文件以tab作为分隔符，但在默认情况下，Hive认为其输入文件的分隔符是ASCII码00（control-A）。

8.8 实践环节：用正确的列分隔符重定义表

下面，我们将修正表规范。

(1) 把下列 HiveQL 语句保存为 commands.hql 文件。

```
DROP TABLE ufodata ;
CREATE TABLE ufodata(sighted string, reported string, sighting_
location string,
shape string, duration string, description string)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' ;
LOAD DATA INPATH '/tmp/ufo.tsv' OVERWRITE INTO TABLE ufodata ;
```

(2) 把数据文件拷贝至 HDFS。

```
$ hadoop fs -put ufo.tsv /tmp/ufo.tsv
```

(3) 执行 HiveQL 脚本：

```
$ hive -f commands.hql
```

该脚本的执行结果如下所示。

```
OK
Time taken: 5.821 seconds
OK
Time taken: 0.248 seconds
Loading data to table default.ufodata
Deleted hdfs://head:9000/user/hive/warehouse/ufodata
```

```
OK
Time taken: 0.285 seconds
```

(4) 验证表中数据的总行数。

```
$ hive -e "select count(*) from ufodata;"
```

该命令的执行结果如下所示。

```
OK
61393
Time taken: 28.077 seconds
```

(5) 验证 reported 列的内容。

```
$ hive -e "select reported from ufodata limit 5"
```

该命令的执行结果如下所示。

```
OK
19951009
19951011
19950103
19950510
19950614
Time taken: 14.852 seconds
```

原理分析

本例中，我们介绍了调用HiveQL命令的第三种方法。除了使用交互shell或在Hive工具中使用查询语句，Hive还可以从包含多条Hive语句的文件中读取其内容，并执行这些命令。

首先，我们创建了这样一个文件，它先删除旧表，然后新建一个表，并把数据导入新表。

新定义的表规约与前一个的主要区别在于，前者用到了ROW FORMAT和FIELDS TERMINATED BY语句。这两条语句都是必需的：第一条命令告诉Hive每行数据包含多个有界字段，而第二条命令则指定了真正的分隔符。可以看出，我们既可以用明确的ASCII码也可以用常用的\t符号来表示tab。



> 在指定分隔符的时候要多加小心，它必须准确无误并且要区分大小写。不要因为大意把\t误写为\T而浪费几个小时的时间，我最近就犯过类似错误。

在运行commands.hql脚本之前，我们再次把UFO数据文件复制到HDFS（上一个文件副本已被DELETE命令删除），然后使用hive -f来执行HiveQL脚本文件。

和之前一样，我们接下来执行两个简单的SELECT语句，第一条语句用于统计表中的总行数，第二条语句用于列出某个指定列的一小部分值。

不出所料，总行数与前一个例子的结果一致。与前一个例子相比，第二条语句的结果看起来是正确的，说明Hive按照每行数据的组成字段进行了正确拆分。

8.8.1　Hive表是个逻辑概念

如果读者仔细观察上个例子中各命令的运行时间，可能会发现一种看似奇怪的现象。把数据导入表所用的时间和创建表规约所用时间基本一样，但是，统计总行数这样的简单任务却用了较长的时间。输出结果也表明，表的创建和数据导入并没有真正引起MapReduce作业的执行，这就解释了为什么执行这些任务所用时间较短。

把数据导入Hive表的过程不同于我们依据传统数据库经验给出的判断。虽然Hive把数据文件拷入工作路径，但事实上它并没有在这个时候将输入数据插入表中各行。与之相反，它以源数据为基础创建了一批元数据，后续HiveQL查询将用到这些元数据。

如此说来，CREATE TABLE和LOAD DATA语句都不会创建实际的表数据，只是生成一些元数据。当Hive使用HiveQL转换成的MapReduce作业访问概念上存储在表中的数据时，将会用到这些元数据。

8.9　实践环节：基于现有文件创建表

截至目前，我们学习了如何把 Hive 有效控制的文件数据直接导入 Hive 表。然而，我们也可以为 Hive 外部文件数据创建表。在需要使用 Hive 处理外部程序写入和管理的数据或数据存储于 Hive 仓库之外的路径的时候，这种方法特别有用。用户不必把这些文件移到 Hive 仓库目录下，用户删除表时也不会影响到这些文件的可用性。

(1) 将以下内容存入 states.sql 脚本文件。

```
CREATE EXTERNAL TABLE states(abbreviation string, full_name
string)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LOCATION '/tmp/states' ;
```

(2) 把数据文件 states.txt 拷贝到 HDFS，然后确认该文件确实存在。

```
$ hadoop fs -put states.txt /tmp/states/states.txt
$ hadoop fs -ls /tmp/states
```

上述命令的执行结果如下所示。

```
Found 1 items
```

```
-rw-r--r--    3 hadoop supergroup    654 2012-03-03 16:54 /tmp/
states/states.txt
```

(3) 执行 HiveQL 脚本。

```
$ hive -f states.hql
```

该命令的执行结果如下所示。

```
Logging initialized using configuration in jar:file:/opt/hive-
0.8.1/lib/hive-common-0.8.1.jar!/hive-log4j.properties
Hive history file=/tmp/hadoop/hive_job_log_
hadoop_201203031655_1132553792.txt
OK
Time taken: 3.954 seconds
OK
Time taken: 0.594 seconds
```

(4) 检查源数据文件。

```
$ hadoop fs -ls /tmp/states
```

该命令的执行结果如下所示。

```
Found 1 items
-rw-r--r--    3 hadoop supergroup    654 ... /tmp/states/states.
txt
```

(5) 对刚创建的表执行一次示例查询。

```
$ hive -e "select full_name from states where abbreviation like
'CA'"
```

该命令的执行结果如下所示。

```
Logging initialized using configuration in jar:file:/opt/hive-
0.8.1/lib/hive-common-0.8.1.jar!/hive-log4j.properties
Hive history file=/tmp/hadoop/hive_job_log_
hadoop_201203031655_410945775.txt
Total MapReduce jobs = 1
... OK
California
Time taken: 15.75 seconds
```

8

原理分析

创建外表的HiveQL语句和之前使用的CREATE TABLE语句格式稍有不同。关键字EXTERNAL表明该表存在于Hive控制之外的位置，同时LOCATION子句指明了源文件或源目录的位置。

在创建了HiveQL脚本之后，我们把源数据文件拷贝至HDFS。我们使用第4章用到的states文

件作为表的源数据，该文件存储了美国州名全称与双字符缩写之间的映射关系。

确认源文件确实存在于HDFS上之后，我们执行查询语句来创建表并再次检查源文件。与上个例子不同，我们没有把源文件移到`/user/hive/warehouse`目录下，`states.txt`文件依然存在于HDFS上的拷贝路径。

最后，我们针对所创建的表执行查询，查询结果验证了表中数据即为源数据。这也突出了本例与`CREATE TABLE`的另一个区别：在上例中非外部表的情况下，创建表的语句不会把数据插入表中，而是由后续`LOAD DATA`或`INSERT`语句（稍后介绍该语句）向表中插入数据。在定义表时用`LOCATION`关键词指定源文件位置，可以在创建表的同时把数据插入表中。

现在，Hive中已有两个表，较大的表用于存储UFO目击事件数据，较小的表用于存储美国州名缩写。如果用第二个表的数据来充实第一个表的location列，会不会很有意义？

8.10　实践环节：执行联结操作

SQL 中经常用到联结这一工具，但在新语言中使用联结似乎不太容易。实质上，联结可以基于某个条件语句实现多表数据行的逻辑组合。Hive 支持多种形式的数据联结，接下来我们将研究这些内容。

(1) 把下列内容存入 `join.hql` 脚本文件。

```
SELECT t1.sighted, t2.full_name
FROM ufodata t1 JOIN states t2
ON (LOWER(t2.abbreviation) = LOWER(SUBSTR( t1.sighting_location,
(LENGTH(t1.sighting_location)-1))))
LIMIT 5 ;
```

(2) 执行上述查询。

```
$ hive -f join.hql
```

该命令的执行结果如下所示。

```
OK
20060930    Alaska
20051018    Alaska
20050707    Alaska
20100112    Alaska
20100625    Alaska
Time taken: 33.255 seconds
```

原理分析

实际上，本例实现的联结查询相对简单。我们想从一系列记录中提取sighted和location字段，

但不想使用location字段的原始数据，而是想把该字段映射为州名全称。我们创建的HiveQL文件执行的就是这个查询任务。HiveQL使用标准的JOIN关键词指定联结语句，并用ON子句指定匹配条件。

由于Hive只支持等联结，也就是说，ON子句中只能进行等式检查，受此限制，事情变得有些复杂。联结语句中的条件子句不能包含>、?、<等操作符，以及我们常用的LIKE关键字。

但是，我们反而有机会介绍一些Hive的内置函数。尤其是，把字符串转换为小写字母的LOWER函数，提取字符串子串的SUBSTR函数，以及返回字符串中字符总数的LENGTH函数。

我们知道，大多数location记录采用了“城市，州名缩写”的形式。所以，要使用SUBSTR提取字符串中倒数第二个和倒数第三个字符，使用length计算字符串长度。由于我们无法保证表中的所有记录都采用了统一的大小写形式，因此还要通过LOWER函数把州名缩写和提取到的字符串都转换成小写形式。

在脚本执行完毕之后，我们发现输出结果与预期一致，它的确包含了目击事件的发生日期，并用州名全称取代了州名缩写。

请注意，LIMIT子句限制了输出的查询结果中包含的行数。这也表明，HiveQL与SQL的某些字词非常相似，这些字词也被MySQL之类的开源数据库所采纳。

本例展示了内部联结的用法。除此之外，Hive还支持左外联结、右外联结和左半联结。如何在Hive中使用联结，其中包含很多微妙之处，如前述等联结限制。如果读者要用到联结，尤其是在非常大的表中使用联结时，应该首先通读位于Hive主页的文档。



我们不是要批判Hive。联结工具的功能非常强大，但公正地说，与其他类型的SQL查询操作相比，编写糟糕的联结或者忽视了某些关键约束条件的联结会更多地导致关系数据库中止运行。

一展身手：使用正则表达式改进联结

除了刚才用到的字符串函数之外，Hive还提供了类似RLIKE和REGEXP_EXTRACT的函数，这些函数支持在Hive中使用类似Java中的正则表达式。重写上例中的联结语句，使用正则表达式进行更准确、更优美的联结操作。

Hive和SQL视图

Hive还支持另一个功能强大的SQL特性——视图。在用户通过SELECT语句指定逻辑表（不是静态表）的内容的时候，视图特别有用，后续的查询语句就可针对这个包含基础数据的动态视图（这也是视图一词的由来）运行。

8.11　实践环节：使用视图

我们可以使用视图隐藏相关的查询复杂性，例如上例中执行的联结操作的复杂性。接下来，我们创建视图实现该功能。

(1) 把下列语句保存为 view.hql 脚本。

```
CREATE VIEW IF NOT EXISTS usa_sightings (sighted, reported,
shape, state)
AS select t1.sighted, t1.reported, t1.shape, t2.full_name
FROM ufodata t1 JOIN states t2
ON (LOWER(t2.abbreviation) = LOWER(substr( t1.sighting_location,
(LENGTH(t1.sighting_location)-1)))) ;
```

(2) 执行 view.hql 脚本。

```
$ hive -f view.hql
```

脚本的运行结果如下所示。

```
Logging initialized using configuration in jar:file:/opt/hive-
0.8.1/lib/hive-common-0.8.1.jar!/hive-log4j.properties
Hive history file=/tmp/hadoop/hive_job_log_
hadoop_201203040557_1017700649.txt
OK
Time taken: 5.135 seconds
```

(3) 再次执行 view.hql 脚本。

```
$ hive -f view.hql
```

脚本的运行结果如下所示。

```
Logging initialized using configuration in jar:file:/opt/hive-
0.8.1/lib/hive-common-0.8.1.jar!/hive-log4j.properties
Hive history file=/tmp/hadoop/hive_job_log_
hadoop_201203040557_851275946.txt
OK
Time taken: 4.828 seconds
```

(4) 针对该视图执行一个测试查询。

```
$ hive -e "select count(state) from usa_sightings where state =
'California'"
```

上述查询语句的执行结果如下所示。

```
Logging initialized using configuration in jar:file:/opt/hive-
0.8.1/lib/hive-common-0.8.1.jar!/hive-log4j.properties
Hive history file=/tmp/hadoop/hive_job_log_
hadoop_201203040558_1729315866.txt
```

```
Total MapReduce jobs = 2
Launching Job 1 out of 2
...
2012-03-04 05:58:12,991 Stage-1 map = 0%,    reduce = 0%
2012-03-04 05:58:16,021 Stage-1 map = 50%,    reduce = 0%
2012-03-04 05:58:18,046 Stage-1 map = 100%,    reduce = 0%
2012-03-04 05:58:24,092 Stage-1 map = 100%,    reduce = 100%
Ended Job = job_201203040432_0027
Launching Job 2 out of 2
...
2012-03-04 05:58:33,650 Stage-2 map = 0%,    reduce = 0%
2012-03-04 05:58:36,673 Stage-2 map = 100%,    reduce = 0%
2012-03-04 05:58:45,730 Stage-2 map = 100%,    reduce = 100%
Ended Job = job_201203040432_0028
MapReduce Jobs Launched:
Job 0: Map: 2    Reduce: 1    HDFS Read: 75416863 HDFS Write: 116
SUCESS
Job 1: Map: 1    Reduce: 1    HDFS Read: 304 HDFS Write: 5 SUCESS
Total MapReduce CPU Time Spent: 0 msec.
OK
7599
Time taken: 47.03 seconds
```

(5) 删除视图。

```
$ hive -e "drop view usa_sightings"
```

该命令的执行结果如下所示。

```
OK
Time taken: 5.298 seconds
```

原理分析

首先，我们使用CREATE VIEW语句创建了一个视图。它与CREATE TABLE类似，但有两个主要区别：

- 列定义中仅包括列名，相关查询会确定各列的数据类型；
- 通过AS子句中指定的SELECT语句生成视图。

我们使用上例中用到的联结语句生成视图，因此，实际上，在创建表的过程中已完成地点字段向州名全称的转换，而没有直接要求用户执行这个标准化过程。

可选的IF NOT EXISTS子句（该子句也可用于CREATE TABLE语句）意味着，如果该视图已经存在的话，Hive会忽视CREATE VIEW语句。如果不使用这个子句，重复创建相同的视图会引发错误，这并不总是我们想执行的操作。

接着，我们执行了两次该脚本来创建视图，同时也验证了使用`IF NOT EXISTS`子句可以防范某些错误，这正是我们所希望的。

成功创建视图之后，我们接下来对它执行一次查询操作。本例中，我们仅统计了发生在加利福尼亚州的UFO目击事件次数。上例中用于生成MapReduce作业的那么多Hive语句现在变成了一行语句，针对该视图的查询需要两个链式MapReduce作业。认证分析上述查询语句和视图定义，就会发现并没什么值得惊奇的。不难想象，上述视图是通过第一个MapReduce作业实现的，它的输出作为下个统计UFO目击事件总数的查询语句的输入，该查询语言的效果与第二个MapReduce作业相同。因此，读者会发现，这个两阶段的作业的执行时间要大于之前任何查询的执行时间。

实际上，Hive的智能程度远高于此。如果视图创建语句中可以包含外部查询的话，Hive只会生成并运行一个MapReduce作业。由于手工开发一系列协同操作的MapReduce作业需要耗费时间，而Hive的一个巨大优势就在于它节省了这些开发时间。尽管用户编写的MapReduce作业的运行效率可能更高，Hive可以在早期帮助用户判定哪个作业是有用的。通过运行一个效率不高的Hive查询就可以判定某个想法是否与预想的一致，而读者耗费一天时间开发MapReduce作业最终无非得到相同结论，我们认为第一个方案稍好一些。

我们曾经提到过，视图会掩盖SQL语句的复杂性，这通常意味着执行视图本来就很费时间。对于大规模生产工作来说，读者不得不优化SQL语句，有时可能需要彻底放弃使用视图。

在查询执行结束之后，我们通过`DROP VIEW`语句删掉了视图，这再次说明HiveQL和SQL在处理表和视图时的相似性。

处理Hive中的畸形数据

细心的读者可能已经发现，上例中通过查询得出的加利福尼亚州发生的UFO目击事件总数与第4章的结果不一致。这是为什么呢?

回想一下，在第4章运行Hadoop Streaming或Java MapReduce之前，我们通过某种方法略去了输入行中的畸形数据。之后在处理数据时，我们使用了更为精确的正则表达式从地点字段提取双字符形式的州名缩写。但是在使用Hive执行相同任务时，我们没有预处理数据，同时采用的提取州名缩写的方法也较为粗糙。

对于后者，我们之前也曾经提到，Hive支持正则表达式，可以使用它来更精细地提取州名缩写。而对前者而言，我们充其量被迫在许多查询语句中加入`WHERE`子句进行复杂的验证。

与上述解决方案不同，常见的模式是在数据导入Hive之前进行数据预处理。例如，在本例中，我们可以运行一个MapReduce作业去掉输入文件中的所有畸形记录，并运行另一个MapReduce作业提前完成地点字段的标准化。

一展身手：实现上述方案

编写一两个MapReduce作业对输入数据进行预处理，生成一个经过净化的更适合直接导入Hive的输入文件。然后编写一个脚本执行上述作业，创建一个Hive表，并把新文件导入这个表。通过上述过程你会发现，使用脚本把Hadoop和Hive整合到一起并不困难，但其功能却非常强大。

8.12　实践环节：导出查询结果

刚才，我们把大量数据导入 Hive 并通过查询语句从中提取了少量数据。我们也可以导出大数据集，下面来看一个例子。

(1) 重新创建刚才用到的视图。

```
$ hive -f view.hql
```

(2) 把下列语句保存为 export.hql 文件。

```
INSERT OVERWRITE DIRECTORY '/tmp/out'
SELECT reported, shape, state
FROM usa_sightings
WHERE state = 'California' ;
```

(3) 执行 export.hql 脚本。

```
$ hive -f export.hql
```

该脚本的执行结果如下所示。

```
2012-03-04 06:20:44,571 Stage-1 map = 100%,    reduce = 100%
Ended Job = job_201203040432_0029
Moving data to: /tmp/out
7599 Rows loaded to /tmp/out
MapReduce Jobs Launched:
Job 0: Map: 2    Reduce: 1    HDFS Read: 75416863 HDFS Write: 210901
SUCESS
Total MapReduce CPU Time Spent: 0 msec
OK
Time taken: 46.669 seconds
```

(4) 查看指定的输出路径。

```
$ hadoop fs -ls /tmp/out
```

结果如下所示。

```
 Found 1 items
-rw-r--r--    3 hadoop supergroup    210901 ... /tmp/out/000000_1
```

(5) 检查输出文件。

```
$ hadoop fs -cat /tmp/out/000000_1 | head
```

结果如下所示。

```
20021014_ light_California
20050224_ other_California
20021001_ egg_California
20030527_ sphere_California
20050813_ light_California
20040701_ other_California
20031007_ light_California
```

原理分析

在重复使用上个视图之后，我们新建了一个HiveQL脚本，该脚本使用了INSERT OVERWRITE DIRECTORY命令。顾名思义，该脚本把后续查询语句的执行结果写入指定位置。OVERWRITE修饰语也是可选的，它指明是否要删除输出目录下的已有内容。跟在INSERT命令后面的SELECT语句生成了要写入输出位置的数据。本例中，我们针对基于联结创建的视图执行查询操作，它说明了查询语句是如此的复杂。

另一个可选的修饰语是LOCAL，如果需要把输出数据写入运行Hive命令的本地文件系统而不是HDFS的话，用户就可选用该修饰语。

在运行上述脚本的时候，MapReduce作业输出的绝大部分内容符合预期，但多了一行内容，它显示的是已向指定输出位置导出的数据总行数。

脚本运行完毕之后，我们转到输出路径，查看结果文件是否保存在该目录下，并检查其内容，结果与预期一致。



因为Hive默认使用ASCII码0001 ('\a')作为输入文本文件的分隔符，所以它在输出文件中也使用ASCII码0001('\a')作为默认分隔符，如上例所示。

用户还可以使用INSERT命令把查询结果插入表中，我们接下来会看到这种例子。但在此之前，我们需要首先解释一个将要用到的概念。

表分区

我们之前曾提到过，在一段较长的时间内，人们对糟糕的联结语句评价很差，因为它会导致

关系数据库耗费大量时间去完成不必要的工作。此外，也会听到关于查询的类似非议，因为查询操作需要执行全表扫描，也就是说，要逐一访问表中的每行数据，而无法使用索引直接访问感兴趣的行。

对于存储在HDFS并映射到Hive表中的数据，一般情况下基本上都依赖于全表扫描。由于无法将数据分割为更有规律的、可直接访问用户感兴趣的数据子集的结构，Hive只能处理整个数据集。对大约为70 MB的UFO文件来讲，问题并不大，因为Hive只用了几十秒就完成了整个文件的处理任务。但是，如果要处理的文件规模是UFO文件大小的1000倍，情况又会如何呢?

就像传统关系数据库一样，Hive可以基于虚拟列的值对表进行分区操作，这些虚拟列的值还会用于后续的查询语句。

特别是，当新建一个表时，用户可指定一列或多列作为分区列，然后在把数据导入表时，这些列的值将决定数据会被写入哪个分区。

对每天都要接收大量数据的表而言，最常用的分区策略就是使用日期列作为分区列。之后我们就可以限制查询语句只处理某个特定分区内的数据。Hive在后台把每个分区的数据存储到自身路径和文件中，这样，它就可以使用MapReduce作业只处理用户感兴趣的数据。通过使用多个分区列，用户可以创建一个多层结构，对于需要频繁查询一小部分数据的大表而言，很有必要花一些时间选择一个最佳的分区策略。

对UFO数据集而言，我们使用目击事件发生的年份作为分区值，但需要使用一些不太常用的特性来使其具有可操作性。因此，在介绍完这部分内容之后，我们接下来要实现一些分区。

8.13　实践环节：制作 UFO 目击事件分区表

接下来，我们将为 UFO 数据新建一个表，以说明表分区的实用性。

(1) 把下列查询语句保存到 createpartition.hql 脚本文件。

```
CREATE TABLE partufo(sighted string, reported string, sighting_
location string,shape string, duration string, description string)
PARTITIONED BY (year string)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' ;
```

(2) 把下列查询语句保存到 insertpartition.hql 脚本文件。

```
SET hive.exec.dynamic.partition=true ;
SET hive.exec.dynamic.partition.mode=nonstrict ;

INSERT OVERWRITE TABLE partufo partition (year)
SELECT sighted, reported, sighting_location, shape, duration,
description,
```

```
SUBSTR(TRIM(sighted), 1,4)    FROM ufodata ;
```

(3) 创建分区表。

```
$ hive -f createpartition.hql
```

上述命令的执行结果如下所示。

```
Logging initialized using configuration in jar:file:/opt/hive-
0.8.1/lib/hive-common-0.8.1.jar!/hive-log4j.properties
Hive history file=/tmp/hadoop/hive_job_log_
hadoop_201203101838_17331656.txt
OK
Time taken: 4.754 seconds
```

(4) 检查刚创建的表。

```
OK
sighted   string
reported  string
sighting_location    string
shape  string
duration  string
description  string
year  string
Time taken: 4.704 seconds
```

(5) 在表中插入数据。

```
$ hive -f insertpartition.hql
```

屏幕上将会显示如下内容。

```
Total MapReduce jobs = 2
...
...
Ended Job = job_201203040432_0041
Ended Job = 994255701, job is filtered out (removed at runtime).
Moving data to: hdfs://head:9000/tmp/hive-hadoop/hive_2012-03-
10_18-38-36_380_1188564613139061024/-ext-10000
Loading data to table default.partufo partition (year=null)
      Loading partition {year=1977}
      Loading partition {year=1880}
      Loading partition {year=1975}
      Loading partition {year=2007}
      Loading partition {year=1957}

...
Table default.partufo stats: [num_partitions: 100, num_files: 100,
num_rows: 0, total_size: 74751215, raw_data_size: 0]
61393 Rows loaded to partufo
...
OK
Time taken: 46.285 seconds
```

(6) 对某个分区数据执行 count 命令。

```
$ hive -e "select count(*)from partufo where year    = '1989'"
```

其结果如下所示。

```
OK
249
Time taken: 26.56 seconds
```

(7) 在未分区表上执行类似查询。

```
$ hive -e "select count(*) from ufodata where sighted like
'1989%'"
```

其结果如下所示。

```
OK
249
Time taken: 28.61 seconds
```

(8) 列出保存分区表的 Hive 目录下的所有文件。

```
$ Hadoop fs -ls /user/hive/warehouse/partufo
```

其结果如下所示。

```
Found 100 items
drwxr-xr-x    - hadoop supergroup        0 2012-03-10 18:38 /
user/hive/warehouse/partufo/year=0000
drwxr-xr-x    - hadoop supergroup        0 2012-03-10 18:38 /
user/hive/warehouse/partufo/year=1400
drwxr-xr-x    - hadoop supergroup        0 2012-03-10 18:38 /
user/hive/warehouse/partufo/year=1762
drwxr-xr-x    - hadoop supergroup        0 2012-03-10 18:38 /
user/hive/warehouse/partufo/year=1790
drwxr-xr-x    - hadoop supergroup        0 2012-03-10 18:38 /
user/hive/warehouse/partufo/year=1860
drwxr-xr-x    - hadoop supergroup        0 2012-03-10 18:38 /
user/hive/warehouse/partufo/year=1864
drwxr-xr-x    - hadoop supergroup        0 2012-03-10 18:38 /
user/hive/warehouse/partufo/year=1865
```

原理分析

本例中，我们创建了两个HiveQL脚本文件。第一个脚本新建了一个分区表。可以看出，它与前面提到的CREATE TABLE语句非常接近，区别在于加入了PARTITIONED BY子句。

在执行完第一个脚本之后，我们检查了表结构，从HiveQL的角度来看，该表很像之前提到

的ufodata表，只是多了一列（year）。这样的话，当在WHERE子句中指定条件时，系统会同样对year列进行处理，即使该列数据并不存在于硬盘数据文件中。

接下来，我们执行第二个脚本，它把数据导入到分区表中。这里需要注意几点。

首先，我们看到INSERT命令可以用于表操作，就像上一节用于目录操作一样。INSERT语句指定了数据的目的位置，跟在INSERT后面的SELECT语句从已有表或视图中提取所需数据。

本节使用的分区方式利用了Hive的一个新功能——动态分区。在大多数情况下，分区子句应包含明确的year列的值。尽管这种方法可以把某一天的数据导入基于日期的分区表，但它不适用于本例中的数据文件类型，因为要所有行的数据插入众多分区表中。通过指定分区列而不设定具体值，Hive会使用SELECT语句返回的year列的值自动生成分区名。

这就解释了为什么会在SELECT语句末尾出现一个奇怪的子句。在指明ufodata表的所有标准列之后，我们加入一个声明，它从sighted列的内容中提取前4个字节。请记住，因为分区表把year分区列视为ufodata表的第7列，这就意味着我们要把每行中sighted字符串中的年度值指定为year列的值。因此，我们在原来每行内容的基础上加上目击事件发生的年份，然后把这些数据插入分区表。

为了证明上述脚本按照预期工作，我们接着进行了两次查询操作。其中一次查询对分区表中1989年的所有记录进行计数，另一次查询对ufodata表中以“1989”字符串开头的记录进行计数。我们曾用“1989”字符串动态生成分区表。

可以看出，这两次查询的结果完全一致，这就证实了我们的分区策略按照预期工作。我们还注意到，对分区表的查询要稍快于对非分区表的查询，尽管二者速度差别并不明显。这可能是因为在处理这种小规模数据集的时候，MapReduce的启动时间在整个作业的运行时间中占据较大的比率。

最后，我们查看了Hive为分区表存储数据的目录，发现该目录下共有100个动态生成的分区表。今后使用HiveQL语句引用某个特定分区，Hive会执行一次意义非凡的优化——它只会处理在相应的分区路径下的数据。

8.13.1　分桶、归并和排序

本节不会详细介绍这些内容，但Hive系统并非只能采用多层分区列这种方法对数据子集的访问过程进行优化。在一个分区中，Hive提供了进一步将数据行聚集到桶中的方法，它通过对CLUSTER BY指定的列使用哈希函数实现。用户还可使用SORT BY对指定列进行排序，经过排序的数据行以有序形式存储在桶中。例如，我们可以基于UFO形状对数据分桶，在每个桶中按目击事件发生的日期进行排序。

读者在第一天使用Hive的时候肯定不会用到这些功能，但是如果用户要处理的数据集越来越大，采用这种优化方式可以显著缩短查询的处理时间。

8.13.2　用户自定义函数

Hive允许用户在HiveQL执行的过程中直接挂接自定义代码。这个功能既可以通过新增库函数实现，也可以通过指定类似于Hadoop Streaming的**Hive transform**实现。本节我们将学习用户自定义函数，添加自定义代码可能是用户在学习Hive的早期阶段最需要的功能。Hive transform是一种较为复杂的机制，用户可用它添加在Hive运行时调用的map和reduce类。如果用户对Hive transform感兴趣的话，Hive wiki对其进行了详细说明。

8.14　实践环节：新增用户自定义函数

接下来，我们演示如何使用 UDF 创建并调用自定义 Java 代码。

(1) 把以下代码保存为 `City.java`。

```
package com.kycorsystems ;

import java.util.regex.Matcher ;
import java.util.regex.Pattern ;
import org.apache.hadoop.hive.ql.exec.UDF ;
import org.apache.hadoop.io.Text ;

public class City extends UDF
{
    private static Pattern pattern = Pattern.compile(
        "[a-zA-z]+?[\\. ]*[a-zA-z]+?[\\, ][^a-zA-Z]") ;

    public Text evaluate( final Text str)
    {
        Text result ;
        String location = str.toString().trim() ;
        Matcher matcher = pattern.matcher(location) ;

        if (matcher.find())
        {
            result = new Text(                    location.
substring(matcher.start(), matcher.end()-2)) ;
        }
        else
        {
            result = new Text("Unknown") ;
        }
        return result ;
    }
}
```

8

(2) 编译 City.java。

```
$ javac -cp hive/lib/hive-exec-0.8.1.jar:hadoop/hadoop-1.0.4-core.
jar    -d . City.java
```

(3) 把生成的类文件打包到 JAR 文件中。

```
$ jar cvf city.jar com
```

上述命令的输出如下所示。

```
added manifest
adding: com/(in = 0) (out= 0)(stored 0%)
adding: com/kycorsystems/(in = 0) (out= 0)(stored 0%)
adding: com/kycorsystems/City.class(in = 1101) (out= 647)(deflated
41%)
```

(4) 启动交互式的 Hive shell。

```
$ hive
```

(5) 把 city.jar 添加到 Hive classpath。

```
hive> add jar city.jar;
```

上述命令的执行结果如下所示:

```
Added city.jar to class path
Added resource: city.jar
```

(6) 确认 city.jar 已添加到 Hive calsspath。

```
hive> list jars;
```

上述命令的执行结果如下所示。

```
file:/opt/hive-0.8.1/lib/hive-builtins-0.8.1.jar
city.jar
```

(7) 为新加入的代码重新注册一个函数名。

```
hive> create temporary function city as 'com.kycorsystems.City' ;
```

上述命令的执行结果如下所示。

```
OK
Time taken: 0.277 seconds
```

(8) 使用新函数执行一次查询。

```
hive> select city(sighting_location), count(*) as total
    > from partufo
    > where year = '1999'
```

```
    > group by city(sighting_location)
    > having total > 15 ;
```

上述命令的执行结果如下所示。

```
Total MapReduce jobs = 1
Launching Job 1 out of 1
...
OK
Chicago    19
Las Vegas    19
Phoenix    19
Portland    17
San Diego    18
Seattle    26
Unknown    34
Time taken: 29.055 seconds
```

原理分析

我们编写的Java类对`org.apache.hadoop.hive.exec.ql.UDF`（User Defined Function）基类进行了扩展。在该类中，我们定义了一个返回指定地点字符串对应的城市名的函数，而地点字符串的模式与前几节完全相同。

实际上，UDF并不会限制**evaluate**函数的参数类型，相反，用户可以自由添加带有任意类型的参数和返回类型的自定义函数。Hive使用Java反射（Reflection）技术选择正确的evaluate函数。如果用户想达到更精细的选择，可以自行开发实现`UDFMethodResolver`接口的实用类。

上例中用到的正则表达式有点难以理解，我们只是想要提取跟在州名缩写后面的城市名，而该正则表达式却如此复杂。然而，城市名的描述方式不一致以及对多个单词组成的名字的处理，造成了上述复杂的正则表达式。除此之外，本例中实现的类较为简单。

接着，我们编译了`City.java`文件，并在此过程中加入一些必需的Hive和Hadoop的JAR文件。

> 请记住，如果用户使用的Hadoop和Hive版本与作者的版本不同，那么hive/lib/hive-exec-0.8.1.jar和hadoop/hadoop-1.0.4-core.jar的文件名可能会有区别。

紧接着，我们把生成的类文件打包成JAR文件，并启动Hive交互式shell。

在创建JAR文件之后，我们需要配置Hive使用该文件。这个过程分两个步骤进行。首先，我们使用`add jar`命令把新建JAR文件添加到Hive使用的classpath。在此之后，我们使用`list jars`

命令确认新JAR文件已经注册到系统中。

添加JAR文件只是告诉Hive存在一些代码，并没有指明我们希望在HiveQL语句中以何种方式引用这些函数。`CREATE FUNCTION`完成的就是这个工作——把Java类与函数名关联起来，本例中，`CREATE FUNCTION`将`city`函数与提供了该函数实现代码的`com.kycorsystems.City`类关联起来。

在把JAR文件添加到classpath，并创建了`city`函数之后，我们可以在HiveQL语句中引用`city()`函数了。

接下来，我们运行一个示例查询，以说明`city()`函数工作正常。回想一下，在UFO目击事件分区表中，我们最希望在哪个地方最常出现UFO，因为所有人都想尽早解开这个千年之谜。

从HiveQL可以看出，我们可以像使用其他函数一样使用新加的用户自定义函数，而且通常只能依靠用户对标准Hive函数库的熟悉程度，来区分哪些函数是内置函数哪些函数是用户自定义函数。

结果显示，美国西北部和西南部集中出现了UFO目击事件，Chicago则是个例外。但是，结果中还包含一些无法确定城市名的数据，我们需要进一步分析其原因，是由于出现UFO的地方不在美国范围内还是正则表达式不够完善需要进一步改进？

8.14.1　是否进行预处理

我们回想一下之前提到的一个话题：是否需要在数据导入Hive之前对其进行预处理，去除畸形数据。从上个例子可以看出，我们可以使用一系列用户自定义函数在进行数据处理的过程中完成类似的数据清洗工作。例如，我们可以加入用户自定义的`state`和`country`函数，从目击事件发生的地点字符串中提取或推导出所在的地区和国家。很难说哪种方法更好一些，但有一些指导原则可以帮助用户决定选用哪种方法。

就像我们用到的例子一样，如果由于种种原因无法处理完整的地点字符串，仅能从中提取几个明显的组成部分，那么可能有必要进行数据预处理。通过预处理我们可以把要访问的列标准化为更可预料的格式，甚至把它分成独立的城市/地区/国家这几列，而不必每次都执行代价较大的文本处理。

但是，如果在HiveQL中经常要用到某一列的原始数据，并且仅在特殊情况下才需要对该列数据进行处理，那么对整个数据集进行代价很大的预处理就没多大好处。

基于上述原则选用对自己的数据和工作任务最有利的处理方案。还要记住，用户自定义函数不仅仅可以执行这种文本处理，它们可以封装用户想对表中数据执行的任何操作代码。

8.14.2 Hive和Pig的对比

在互联网上搜索关于Hive的文章时，经常会发现人们往往将Hive和另一个称为**Pig**的Apache项目进行对比。其中关于它们之间对比的最常见问题是：为什么二者都存在，什么时候该选用哪一个工具，哪个工具更好一些，以及使用哪种工具显得更酷一些。

二者的相似之处在于，Hive提供了一种类似SQL语言的接口用于数据处理，而Pig用到了一种叫做**Pig Latin**的语言定义数据流流水线。就像Hive先把HiveQL翻译成MapReduce，然后执行这些MapReduce作业一样，Pig会执行类似的代码生成过程，它把Pig Latin脚本翻译为MapReduce代码。

二者的最大区别在于对作业执行方式的控制粒度。HiveQL就像SQL一样，它只定义要执行的操作，却几乎不管如何实现这些操作。HiveQL查询规划器负责安排HiveQL命令的执行顺序，evaluate函数的执行顺序等。Hive在运行时给出上述安排，它的工作模式类似于传统关系数据库的查询规划器。Pig Latin也是在查询规划器这一层操作数据。

使用Hive和Pig都避免了直接编写MapReduce代码，只不过两种方法的抽象方式不尽相同。

到底是选用Hive还是选用Pig最终取决于用户需求。如果用户更希望使用熟悉的SQL接口操作数据，它可以使Hadoop中的数据用于更广泛的受众，那么很明显应当选用Hive。但如果有专门人员以数据流水线的方式考虑问题，并需要对作业运行方式进行更细粒度的控制，那么可能Pig会是一个更好的选择。Hive和Pig项目都在寻求进一步整合，因此，关于它们属于竞争关系的错误观念会得到改善。二者作为相互补充的技术，都可降低执行MapReduce作业的门槛。

8.14.3 未提到的内容

在本章对Hive的综述中，我们介绍了它的安装和配置方法，如何创建表并在表中插入数据，怎样创建视图以及如何实现联结等内容。我们演示了如何把数据导入、导出Hive，如何优化数据处理过程，并研究了几个Hive内置函数。

实际上，我们仅学习了Hive的皮毛。我们不仅没有深入研究上述几个话题和相关概念，甚至都没有接触类似**MetaStore**和**SerDe**的内容。Hive将其配置和元数据存储在MetaStore中，而SerDe（serialize/deserialize）则用于从JSON这样的复杂文件格式读取数据。

Hive是一种功能非常丰富的工具，它提供了很多复杂而又强大的功能。如果读者认为Hive对自己非常有用，那么建议读者在学完本章例子之后，花时间阅读Hive网站上的文档。你还会在那里发现用户邮件列表，它是一个很好的信息来源，通过它可以获得别人的帮助。

8.15　基于 Amazon Web Services 的 Hive

弹性MapReduce对Hive的支持力度很大，它提供了一些特殊机制，有助于把Hive整合到其他AWS服务中。

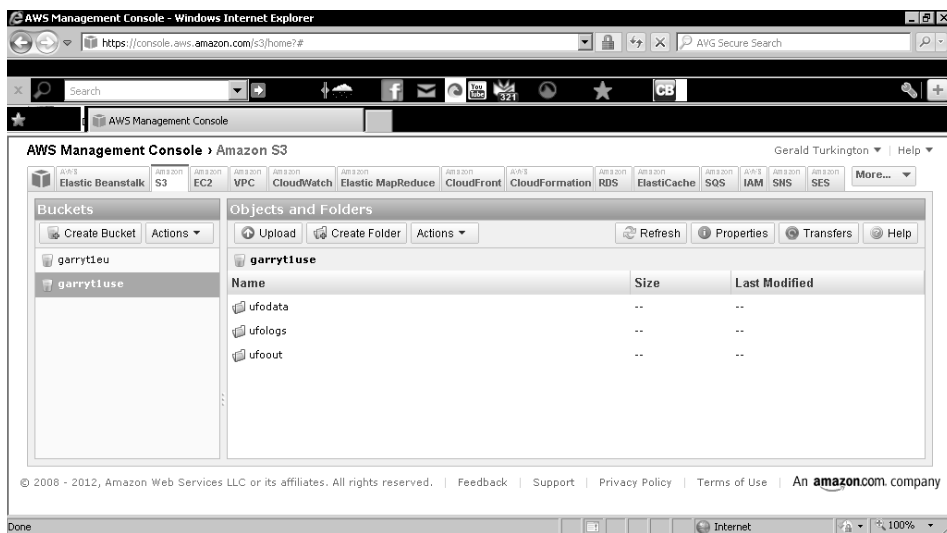
8.16　实践环节：在 EMR 上分析 UFO 数据

接下来，我们将通过在 EMR 平台上分析 UFO 数据来研究如何在 EMR 环境中使用 Hive。

(1) 登录 AWS 管理控制台，其地址为 http://aws.amazon.com/console。

(2) EMR 上的每个 Hive 作业流都从一个 S3 桶开始运行，我们需要选择用作存储源数据的桶。选择 **S3** 以查看用户账户创建的桶列表，之后从中选择一个桶作为作业流的数据来源。在本例中，我们选用名为 garryt1use 的桶。

(3) 通过网页接口在 garryt1use 桶中新建 3 个目录，它们分别是 `ufodata`、`ufoout` 和 `ufologs`。桶中内容的列表如下图所示：



(4) 双击并打开 `ufodata` 目录，在该目录下创建两个名为 `ufo` 和 `states` 的子目录。

(5) 把下列代码保存为 `s3test.hql` 脚本，点击 `ufodata` 目录中的 **Upload** 链接，按照提示上传该脚本文件。

```
CREATE EXTERNAL TABLE IF NOT EXISTS ufodata(sighted string,
reported string, sighting_location string,
```

```
shape string, duration string, description string)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LOCATION '${INPUT}/ufo' ;

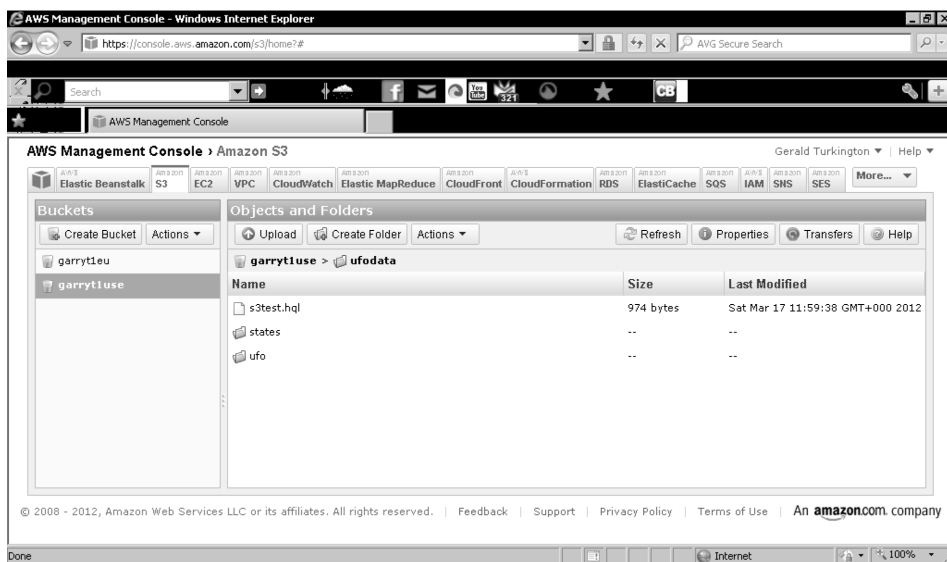
CREATE EXTERNAL TABLE IF NOT EXISTS states(abbreviation string,
full_name string)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LOCATION '${INPUT}/states' ;

CREATE VIEW IF NOT EXISTS usa_sightings (sighted, reported, shape,
state)
AS SELECT t1.sighted, t1.reported, t1.shape, t2.full_name
FROM ufodata t1 JOIN states t2
ON (LOWER(t2.abbreviation) = LOWER(SUBSTR( t1.sighting_location,
(LENGTH(t1.sighting_location)-1)))) ;

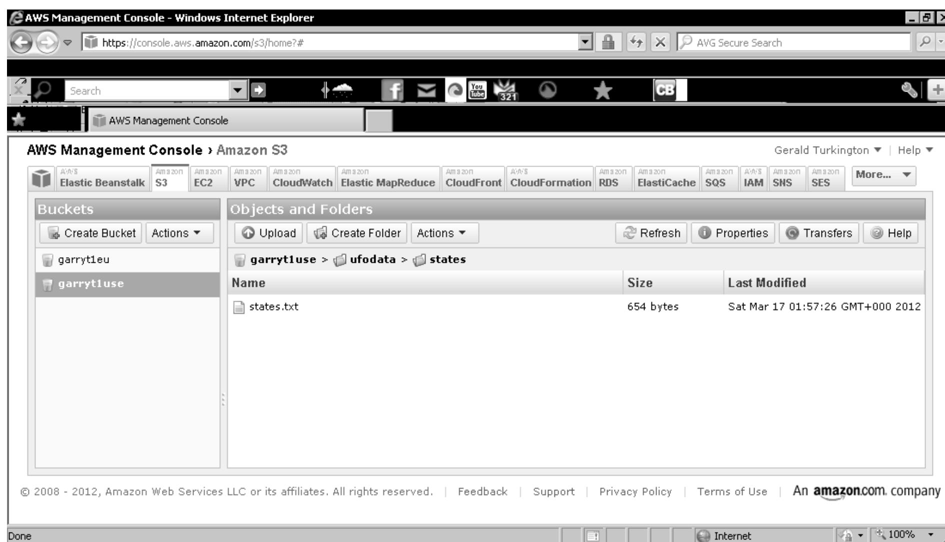
CREATE EXTERNAL TABLE IF NOT EXISTS state_results ( reported
string, shape string, state string)
ROW FORMAT DELIMITED
FFIELDS TERMINATED BY '\t' LINES TERMINATED BY '\n'
STORED AS TEXTFILE
LOCATION '${OUTPUT}/states' ;

INSERT OVERWRITE TABLE state_results
SELECT reported, shape, state
FROM usa_sightings
WHERE state = 'California' ;
```

Ufodata 目录中的文件列表如下图所示。

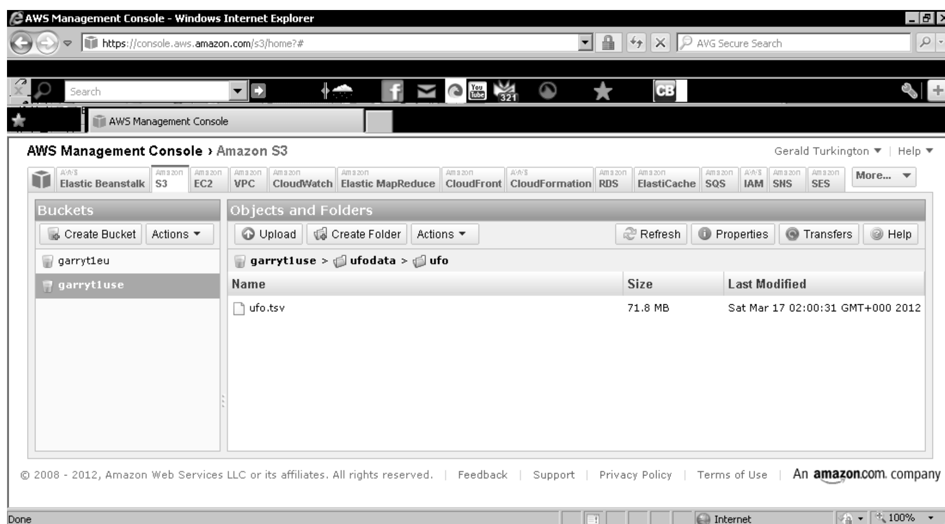


(6) 双击并打开 states 目录，并把之前曾用到的 states.txt 文件上传到该目录。该目录中的文件列表如下图所示。

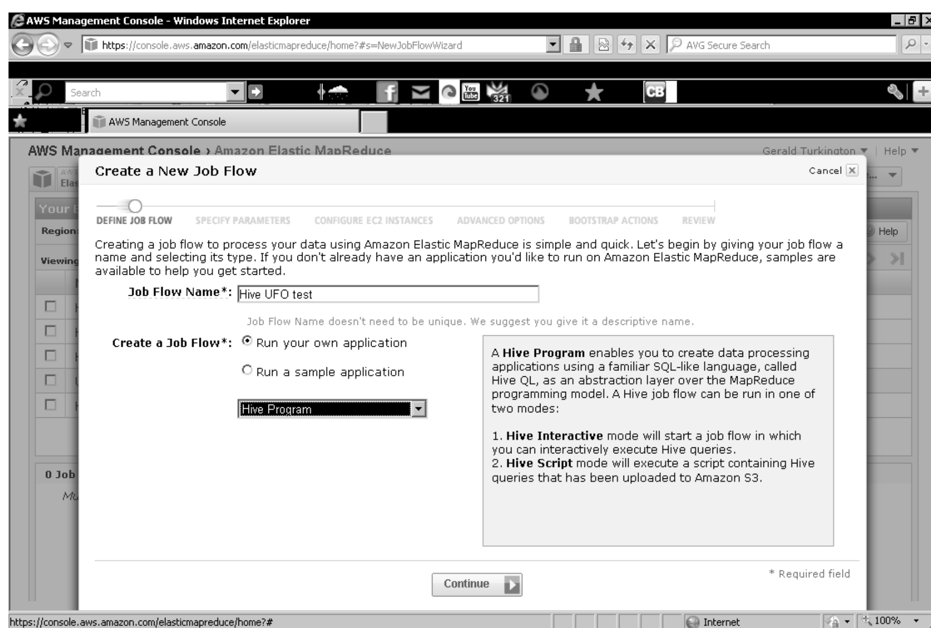


(7) 点击文件列表顶部的 ufodata 部件，然后返回该目录。

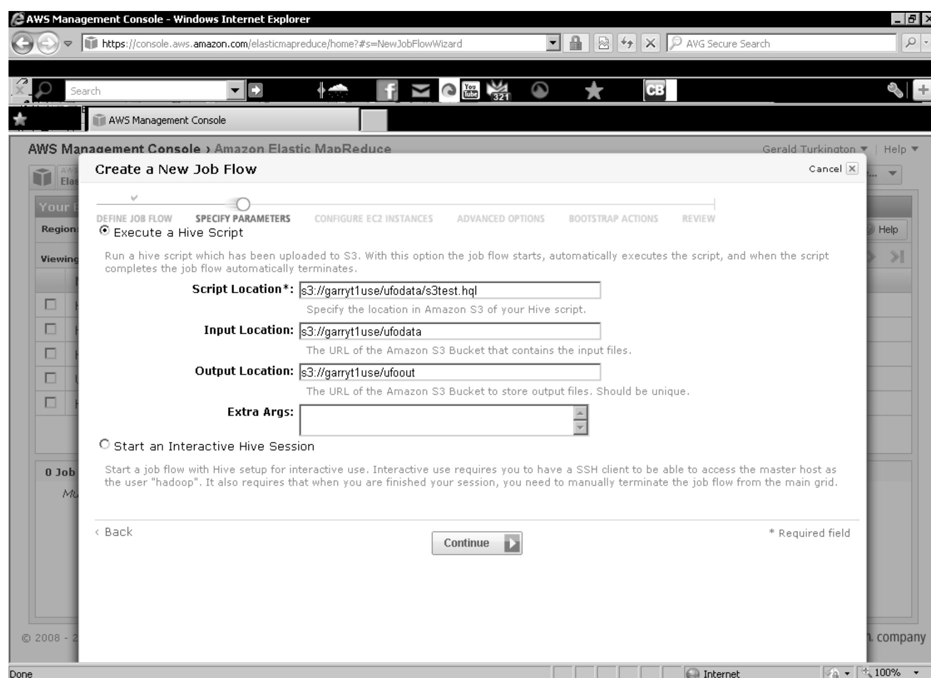
(8) 双击并打开 ufo 目录，把之前曾用到的 ufo.tsv 文件上传到该目录。该路径下的文件列表如下图所示。



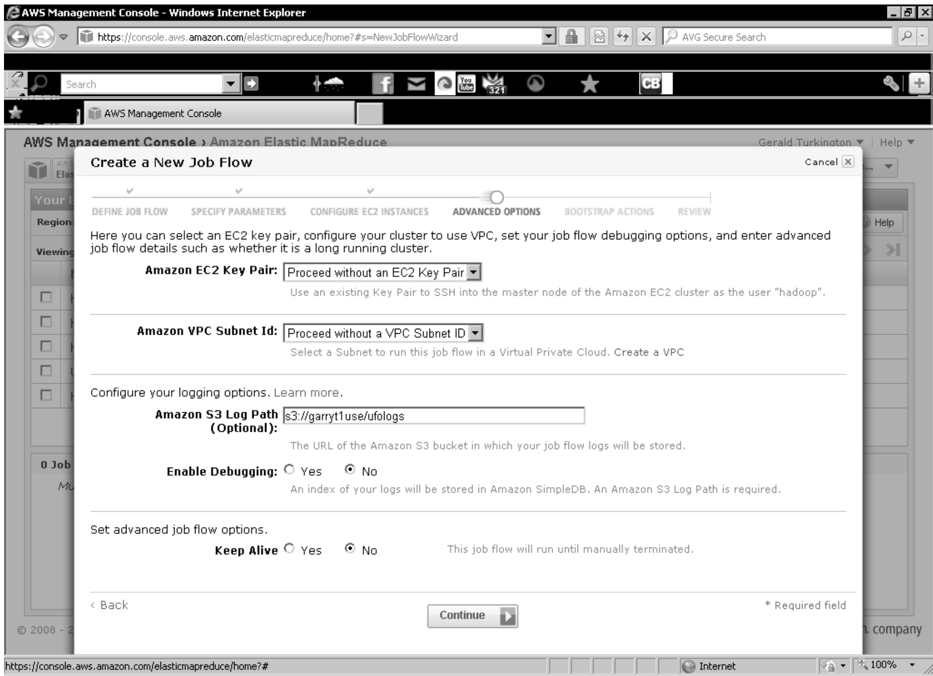
(9) 现在，选择 **Elastic MapReduce** 并点击 **Create a New Job Flow**。然后选择 **Run your own application**，并选择 Hive Program，如下图所示。



(10) 点击 **Continue** 按钮，之后填写 Hive 作业流所需的细节信息。以下图为例，但要记住把桶名（`s3://URLs` 的第一部分）改为用户刚刚设置的桶。



(11) 点击 **Continue** 按钮，检查要使用的主机类型和主机数量，然后再次点击 **Continue** 按钮。然后填写存放日志文件的目录名，如下图所示。



(12) 单击 **Continue** 按钮。之后在剩余的作业创建步骤中一直点击 Continue 按钮，因为用户无需修改剩余步骤中的默认设置。最后，启动作业流并通过管理控制台查看其进度。

(13) 一旦作业成功完成，返回 **S3** 并双击 `ufoout` 目录。该目录下应该有一个 `states` 目录，在该目录中有一个名为 `0000000` 的文件。双击并下载该文件，验证其内容如下所示。

```
20021014    light    California
20050224    other    California
20021001    egg      California
20030527    sphere   California
```

原理分析

在实际执行EMR作业流之前，我们需要完成一些设置工作。首先，我们使用S3的网页接口为作业创建目录结构。我们创建了3个主目录，分别用于存储输入数据，写入作业结果，以及存储EMR在执行作业流时产生的日志。

HiveQL脚本中包含一些本章前几节用到的Hive命令，本例中对它们做了一些修改。我们为UFO目击事件数据和州名创建了两个表，并创建了一个联结这两个表的视图。接着，我们创建了一个没有源数据的新表，并通过`INSERT OVERWRITE TABLE`语句把一个查询的结果数据插入该表。

该脚本的特别之处在于为每个表指定`LOCATION`子句的方式。对存储输入数据的表而言，我们使用相对于INPUT变量的路径；类似地，我们使用相对于`OUTPUT`变量的路径作为存储结果数据的表的输出位置。

请注意，EMR中的Hive希望表数据的位置是一个路径而非文件。这也就解释了之前我们为什么要为每个表都创建一个子目录，然后把特定源文件上传到相应目录中，而不是直接指定每个表要用的数据文件的路径。

在S3桶中创建了必需的文件和目录结构之后，我们转向EMR网页控制台，并开始创建作业流。

在指明我们希望使用自有Hive程序之后，我们在一个页面中填入了作业流需要的一些关键数据：

- HiveQL脚本本身的位置；
- 输入数据所在路径；
- 写入输出数据的目录。

HiveQL脚本的自身路径是一个很明确的路径，无需对它进行任何解释。但是，输入数据路径和输出数据路径是如何映射为Hive脚本中使用的`INPUT`和`OUTPUT`变量的，理解这一点非常重要。

Hive脚本使用`INPUT`变量指代输入路径，这就是我们把包含UFO目击事件数据的路径写为`${INPUT}/ufo`的原因。类似地，Hive脚本使用`OUTPUT`变量指代输出路径。

我们没有对默认的主机设置进行任何改动，选用了一台较小的主节点主机和两台较小的核心节点主机。在下一个页面中，我们添加了作业流运行过程中产生的日志的存储位置。

尽管用户可选择是否输出日志，但捕获这些日志是有用的，尤其是在运行新脚本的早期阶段，虽然在S3服务中存储这些日志信息是要付费的。EMR还可以把有索引的日志数据写入另一个AWS服务——**SimpleDB**，但我们没有演示其使用方式。

在完成作业流定义之后，我们启动该作业流。在作业流成功执行后，我们转向S3接口浏览输出目录，不出所料，该目录中包含着作业流的输出结果。

8.16.1　在开发过程中使用交互式作业流

在开发用于EMR的新Hive脚本时，上节讲的按批执行作业的方式不太合适。通常在作业流

创建和执行过程中有几分钟的延迟，如果作业失败的话，会浪费EC2实例几个小时的开销。

与上例中选择不同的选项创建作业流来运行Hive脚本不同，我们还可以以交互式模式启动Hive作业流。它可以加快Hadoop集群的设置，却不需要脚本。你可以通过SSH方式以集群用户的身份登录到安装有Hive的主节点。在这样的环境中开发脚本更有效率。如果需要的话，把作业流脚本设置为自动执行方式。

一展身手：使用交互式的EMR集群

在EMR中启动一个交互式的Hive作业流。你会用到已在EC2中注册好的SSH证书来连接到主节点。直接在主节点上运行上个脚本，记得要为脚本提供合适的参数。

8.16.2 与其他AWS产品的集成

在本地安装Hadoop/Hive之后，数据的存储位置无非就是HDFS或者本地文件系统。从前几节内容可以看出，EMR中的Hive提供了另外一种选择，它支持从外部表中获取数据，例如S3。

另一种AWS服务也有类似的功能，它就是**DynamoDB**（网址为http://aws.amazon.com/dynamodb）。它是托管在云端的NoSQL数据库。运行于EMR的Hive作业流可以声明使用外部表，既可以从DynamoDB读取数据，也可以将其作为查询结果的输出位置。

这个模型功能非常强大，因为在这种模式下，用户可以使用Hive来处理和合并多个数据源的数据，而从某个系统向Hive表的数据映射机制是透明的。另外，Hive还可以借助DynamoDB把数据从一个系统迁移到另一个系统。采用这种数据存储方案的主要障碍在于，需要频繁地将存储在HDFS或本地文件系统的数据转移到托管的云存储系统中。

8.17 小结

本章学习了Hive的相关知识，用过关系数据库的读者应该非常熟悉Hive提供的许多工具和功能。由于避免了开发MapReduce程序，Hive能够使更多的人感受到Hadoop的强大之处。

特别是，我们下载并安装了Hive，了解到它是一个把HiveQL语言转换成MapReduce代码的客户端程序，然后再把MapReduce程序提交给Hadoop集群。我们研究了Hive创建表并针对这些表执行查询的原理。我们观察到，Hive可以支持多种基础的数据文件格式和数据结构，并学习了如何修改相应的设置选项。

通过本章学习，我们还认识到，Hive表在很大程度上是一个逻辑概念。有了这种认识之后，所有针对表的类似SQL的操作事实上都是由MapReduce作业在HDFS文件上完成的。之后，我们

学习了如何在Hive中使用联结和视图，以及如何对表进行分区以辅助高效查询操作。

我们使用Hive把查询结果写入HDFS上的文件，并学习了如何在弹性MapReduce中使用Hive，怎样使用交互式作业流开发新的Hive程序，并在批运行模式中自动运行这些程序。

我们在本书中曾多次提到，Hive看上去是个关系数据库，其实并非如此。但是，在很多情况下，读者需要整合某些现有的关系数据库。下一章将会介绍如何实现这种整合，以及如何实现在不同类型数据源之间移动数据。

第9章 与关系数据库协同工作

从上一章可以看出，Hive的功能非常强大。用户可以把存储在Hadoop中的数据近似看做关系数据库。然而，它终究不是一个真正的关系数据库。它没有完全实现SQL标准，它的性能和规模特征与传统关系数据库区别很大（并不是说哪个更好一些）。

很多情况下，读者会发现Hadoop集群需要与关系数据库配合完成某些数据处理任务。通常，业务流需要把数据从一个存储系统移动到另一个存储系统。本章，我们将研究如何在不同存储系统之间转移数据。

本章包括以下内容：

- 介绍一些常见的Hadoop/RDBMS使用案例；
- 学习如何将数据从RDBMS移至HDFS和Hive；
- 最好使用Sqoop解决上述问题；
- 使用Sqoop将Hadoop数据导出至RDBMS；
- 最后讨论如何在AWS中使用Sqoop。

9.1 常见数据路径

早在第1章我们就曾讨论过在什么情况下使用Hadoop，在什么情况下使用传统关系数据库。和当时的解释一样，我们认为要根据手头的具体任务选择合适的工具，甚至还有可能要同时用到多种技术。为阐明这一观点，我们来看一些具体的例子。

9.1.1 Hadoop用于存储档案

把RDBMS用作主要的数据储存库时，经常会遇到数据规模和数据保留方面的问题。随着新数据量的增长，该如何处理那些价值不太大的老旧数据呢？

习惯上，有两种主要的解决方案。

- 对RDBMS进行分区，这样会提高较新数据的访问性能。有时，可以使用速度较慢、成本较低的存储系统储存老旧数据。
- 使用磁带或其他线下存储设备存储老旧数据。

这是两种有效的解决办法。到底选用哪种方法取决于是否需要实时访问这些老旧数据。这两种方法都比较极端：第一种方法保证了访问速度的最大化，却提高了系统复杂性并增加了设备成本；第二种方法虽然降低了成本但无法实时访问数据。

最近出现了一种新的解决方案：用关系数据库存储最新数据，同时使用Hadoop存储老数据。采用这种方案后，数据要么以结构化文件的形式存储在HDFS上，要么存储在Hive并保留了RDBMS接口。这是一种两全其美的好办法，用户既可以通过高速、低延迟的SQL查询访问数据规模较小的新数据，也可以通过Hadoop访问数据规模较大的存档文件。因此，无论用户通过哪种方式，都能访问目标数据。既支持新数据查询，也支持存档数据查询的存储平台尤其需要采用这种方案。

由于Hadoop的扩展性极强，这种模型具有很好的成长潜力。有了它，我们可以持续不断地增加存档数据的容量，但同时也能够对其进行分析。

9.1.2 使用Hadoop进行数据预处理

在讨论Hive的时候我们曾多次强调，有些情况下运行预处理作业或以其他方式净化数据是非常有用的。但不幸的是，在大多数大数据处理案例中，大量数据来自于多个数据源，这就意味着这些数据中间必然存在错误数据。尽管大多数MapReduce作业只会处理其中部分数据，但我们仍希望找出全部的不完整数据或错误数据。最好先预处理数据，然后再把它们存入Hive，传统的关系数据库同样如此。

Hadoop非常适合完成这个任务，它从多个数据源获取数据，进行必要的转换之后把它们合并，然后在数据插入关系数据库之前完成数据净化工作。

9.1.3 使用Hadoop作为数据输入工具

Hadoop不仅能净化数据，使其更适合导入关系数据库。而且，它还可以用于生成其他数据集或数据视图，然后在关系数据库中使用这些结果。举个例子，常见的应用场景是，我们不仅希望显示某个账号的主要数据，也想同时显示根据账号使用情况生成的二级数据。其实就是对前几个月不同消费类型的交易进行汇总。这些数据存储在Hadoop中，基于这些数据可以生成实际汇总，把它们也存储在数据库中以便快速查询。

9.1.4 数据循环

实际情况往往要比这些单一的场景复杂得多。通常，Hadoop和关系数据库之间的数据流是环形或弧形的，而不是简单的从Hadoop到关系数据库或从关系数据库到Hadoop的线性路径。例如，数据可能先经过Hadoop的预处理，然后存储在关系数据库中，接着又频频与某些运算的结果聚合，再把相应的结果存入数据库。某些不太常用的数据符合一定的标准之后，用户就会从数据库中删除这些数据，但还要在Hadoop上存档。

先不考虑这些复杂的情况。在把Hadoop集成到已有IT系统的时候，保证数据在Hadoop和关系数据库之间的流动能力至关重要。接下来，我们将学习具体的实现方法。

9.2 配置 MySQL

在向关系数据库进行读写操作之前，我们首先需要一个正在运行的关系数据库。由于MySQL数据库可免费使用且应用范围极广，得到了许多开发者的青睐，因此本章使用MySQL数据库作为关系数据库的代表。当然，读者可以选用JDBC驱动支持的任何关系数据库，但如果读者选用了MySQL之外的其他关系数据库，在与数据库服务器建立连接时需要作出相应的调整。

9.3 实践环节：安装并设置 MySQL

本节将学习如何安装并配置 MySQL，赋予用户基本的数据库操作权限和访问权限。

(1) 在 Ubuntu 主机上，使用 `apt-get` 命令安装 MySQL。

```
$ apt-get update
$ apt-get install mysql-server
```

(2) 根据提示，为 root 用户设置一个合适的密码。

(3) 安装完成后，连接到 MySQL 服务器。

```
$ mysql -h localhost -u root -p
```

(4) 按照提示，输入 root 密码。

```
Welcome to the MySQL monitor.    Commands end with ; or \g.
Your MySQL connection id is 40
...
Mysql>
```

(5) 新建一个数据库，用于演示本章讲到的例子。

```
Mysql> create database hadooptest;
```

上述命令的执行结果如下。

```
Query OK, 1 row affected (0.00 sec)
```

(6) 新建一个用户账户，并赋予该用户全部数据库权限。

```
Mysql>    grant all on hadooptest.* to 'hadoopuser'@'%' identified
by 'password';
```

上述命令的执行结果如下。

```
Query OK, 0 rows affected (0.01 sec)
```

(7) 重新加载用户权限，使上一步的配置生效。

```
Mysql> flush privileges;
```

上述命令的执行结果如下。

```
Query OK, 0 rows affected (0.01 sec)
```

(8) 退出 root 用户账号。

```
mysql> quit;
```

上述命令的执行结果如下：

```
Bye
```

(9) 使用 hadoopuser 账号登录，根据提示输入密码。

```
$ mysql -u hadoopuser -p
```

(10) 切换到新建数据库 hadooptest。

```
mysql> use hadooptest;
```

(11) 创建一个测试表，然后删掉该表以验证该用户具备确实相应的操作权限，随后退出。

```
mysql> create table tabletest(id int);
mysql> drop table tabletest;
mysql> quit;
```

```
garry@vm16:~$ mysql -u hadoopuser -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 44
Server version: 5.1.66-rel14.1 (Percona Server (GPL), 14.1, Revision 495)

Copyright (c) 2000, 2012, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use hadooptest
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> create table  tabletest(id int) ;
Query OK, 0 rows affected (0.01 sec)

mysql> drop table tabletest;
Query OK, 0 rows affected (0.00 sec)

mysql> quit;
Bye
garry@vm16:~$
```

原理分析

多亏apt软件包管理器的神奇作用，我们可以非常轻松地安装像MySQL这样的复杂软件。我们使用的是在Linux系统上安装软件的标准流程。在Ubuntu（以及许多其他Linux版本）系统上，请求下载MySQL的服务器软件包会同时把它所依赖的所有软件包都下载到客户端主机，包括MySQL的客户端安装包。

在安装过程中，系统会提示用户输入数据库的root口令。即使该数据库仅作实验之用，用户不会在该数据库中存储任何有价值的数据，仍有必要为root用户设置一个强口令。为root用户设置弱口令的习惯非常糟糕，我们坚决反对这种行为。

安装好MySQL之后，我们使用mysql命令行工具连接到数据库。该命令有多个选项，但我们仅会用到以下几个。

- -h：该选项用于指定要连接的数据库的主机名（缺省状态下指的是本地主机）；
- -u：该选项用于指定使用哪个用户账号连接数据库（默认情况下指的是当前Linux用户）；
- -p：该选项用于指定用户口令。

MySQL支持多数据库，每个数据库都是一组表的集合，同时每个表都必须从属于某个数据库。MySQL已经内置了几个数据库，但我们要新建一个测试用的数据库，因此我们使用CREATE DATABASE语句新建了一个名为hadooptest的数据库。

除非用户被明确赋予执行所需操作的权限，否则MySQL不会执行任何数据库操作。我们不想以root用户的身份完成所有任务（使用root权限进行所有数据库操作不仅是一个坏习惯，同时也存在巨大的风险，因为root用户可以修改或删除任何数据），因此我们使用GRANT语句新建了一个名为hadoopuser的用户。

我们使用GRANT语句完成以下3个任务：

- 创建hadoopuser账号；
- 为hadoopuser用户设置口令。虽然本例中我们使用password作为该用户的口令，但用户千万不要这样做，而应当选用容易记住的字符组合；
- 赋予hadoopuser用户全部权限，使其可以对hadooptest数据库及组成该数据库的所有数据表执行任何操作。

我们通过FLUSH PRIVILEGES命令确保这些设置生效，然后退出root账号，并以hadoopuser的身份连接到服务器，查看数据库工作是否正常。

本例中的USE语句有些多余。以后，我们可以在mysql命令行工作中加入要访问的数据库名，这样就会自动切换到那个数据库。

使用新用户成功连接到数据库是个好兆头，但为了确信所有设置均已生效，我们在hadooptest数据库中新建一个表，然后删除该表。上述操作的成功表明hadoop用户确实拥有修改数据库的权限。

小心翼翼的原因

我们是不是有点太过谨慎了，非要仔细检查MySQL的每一步设置。但是，我曾经发现，某些细微的拼写错误，尤其是在GRANT语句中，都会导致一些很难发现的问题。为了确保不出错，我们接下来要修改MySQL的默认配置，其实我们并不需要这么做，但如果不这样做，将来可能会后悔。

在产品数据库中，用户当然不能像书中这样随意使用与数据库安全性密切相关的语句，例如GRANT。一定要查阅数据库文档，完全弄清楚用户账号及数据库权限的相关内容。

9.4　实践环节：配置 MySQL 允许远程连接

MySQL 的默认设置会拒绝其他主机访问数据库，我们需要对此进行修改。

(1) 使用用户最喜欢的文本编辑器编辑 /etc/mysql/my.cnf，找到下面这行内容。

```
bind-address = 127.0.0.1
```

(2) 在该行前面加入注释符“#”。

```
# bind-address = 127.0.0.1
```

(3) 重启 MySQL。

```
$ restart mysql
```

原理分析

大多数MySQL的默认配置只允许从本地主机访问数据库。从安全的角度来看，这个配置无疑是非常正确的。但是，它也会引起一些现实问题，例如，用户使用的MapReduce作业需要访问远程数据库。用户会发现由于无法连接数据库，作业最终失败。在这种情况下，用户在MySQL主机上启动mysql命令行客户端，运行成功，没有发现任何问题。之后，用户可能编写一个用于测试连通性的JDBC客户端程序。同样，也没有发现问题。只有当Hadoop工作节点要连接MySQL服务器时才会出现上述问题。这种情况同样曾困扰了我一段时间！

上述对MySQL默认设置的修改会把MySQL绑定到所有可用接口，因此，可以从远程客户端访问MySQL数据库。

在修改默认配置后，需要重新启动MySQL服务器。在Ubuntu 11.10系统中，许多服务脚本迁移到了Upstart框架，我们可以直接手动运行restart命令。

如果用户使用的操作系统不是Ubuntu，或是Ubuntu的其他版本，MySQL配置文件在主机上的存储位置可能稍有不同。例如，在CentOS和Red Hat企业版操作系统上，MySQL配置文件存放在/etc/my.cnf目录下。

禁止尝试的一些操作

至少要考虑后果。在前几个例子中，我们为新建用户帐号设置了弱口令，千万别这么做。特别是在数据库支持远程访问时，千万不能设置弱口令。不错，这只是一个测试数据库，其数据没有任何价值，但令人惊讶的是，很多测试数据库的使用时间都很长，并且越来越关键。在此情况下，用户是否会记得删除配置了弱口令的用户账号？

讲得够多了。数据库中需要数据，接下来，我们将在hadooptest数据库中新建一个数据表，本章其余几节也会用到这个表。

9.5 实践环节：建立员工数据库

大多数数据库教程都使用员工数据表作为示例。可以说，如果不讨论员工数据库，那么数据库的学习就不完整。因此，我们按照惯例，在 hadooptest 数据库中新建员工数据表。

(1) 新建 employees.tsv 文件，它以 tab 键为分隔符。其内容如下。

```
Alice	Engineering	50000	2009-03-12
Bob	Sales	35000	2011-10-01
Camille	Marketing	40000	2003-04-20
David	Executive	75000	2001-03-20
Erica	Support	34000	2011-07-07
```

(2) 连接 MySQL 服务器。

```
$ mysql -u hadoopuser -p hadooptest
```

(3) 创建数据表。

```
Mysql> create table employees(
first_name varchar(10) primary key,
dept varchar(15),
salary int,
start_date date
) ;
```

(4) 把 employees.tsv 文件数据导入数据表：

```
mysql> load data local infile '/home/garry/employees.tsv'
    -> into table employees
    -> fields terminated by '\t' lines terminated by '\n' ;
```

```
garry@vm16: ~
File Edit View Terminal Help
Server version: 5.1.66-rel14.1 (Percona Server (GPL), 14.1, Revision 495)

Copyright (c) 2000, 2012, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> create table employees(
    -> first_name varchar(10) primary key,
    -> dept varchar(15),
    -> salary int,
    -> start_date date);
Query OK, 0 rows affected (0.02 sec)

mysql> load data local infile '/home/garry/employees.tsv'
    -> into table employees
    -> fields terminated by '\t' lines terminated by '\n';
Query OK, 5 rows affected (0.01 sec)
Records: 5  Deleted: 0  Skipped: 0  Warnings: 0

mysql>
```

原理分析

这是一个标准的数据库操作流程。我们创建了一个以tab键为分隔符的数据文件，在数据库中创建数据表，然后使用LOAD DATA LOCAL INFILE语句将文件数据导入数据表。

因为我们的目的只是为了说明如何把数据导入MySQL数据库，因此本例中使用了一个很小的数据集。

留意数据文件的访问权限

别忽视了LOAD DATA语句中的LOCAL关键字。这样做，MySQL会以MySQL用户的身份导入数据文件，通常会引起文件访问问题。

9.6　把数据导入 Hadoop

我们前期已做了大量努力，接下来，我们看看如何从MySQL数据库导出数据，并导入Hadoop。

9.6.1　使用MySQL工具手工导入

把MySQL的导出数据导入Hadoop的最简单方法就是使用现有的命令行工具和MySQL语句。为了导出整个数据表或整个数据库的内容，MySQL提供了mysqldump工具。为了进行更精确的数据导出，我们可以使用下述SELECT语句。

```
SELECT col1, col2 from table
INTO OUTFILE '/tmp/out.csv'
FIELDS TERMINATED by ',', LINES TERMINATED BY '\n';
```

一旦我们把数据导出到文件中，就可以使用hadoop fs -put把该文件移到HDFS上，或通过上一章讲过的方法将其导入Hive。

一展身手：把员工数据表的内容导出至HDFS

我们并不想把本章变成MySQL教程。请自行查看mysqldump工具的语法，然后用它或SELECT ... INTO OUTFILE语句把员工数据表导出到以tab为分隔符的文件，随后将该文件拷贝至HDFS。

9.6.2　在mapper中访问数据库

对我们的小例子而言，上述方法确实不错。但如果读者需要导出一个更大的数据集，尤其是

要用MapReduce作业处理导出数据的时候，有什么更好的办法吗？

一种直观的方法是，在MapReduce输入作业中直接使用JDBC从数据库读取数据并写入HDFS，为后续的数据处理做好准备。

这种方法虽然有效，却存在一些不太明显的问题。

读者需要仔细考虑数据库可以承载的负荷。在一个规模很大的集群上运行这种作业会迅速拖垮数据库，因为成千上万个mapper要同时与数据库建立连接并读取同一个数据表的内容。最简单的访问模式可能是每次读取一行数据，这种方法的效率不及成块访问方式的效率高。即使数据库能够承载这么大的访问量，数据库的网络连接也会马上成为瓶颈问题。

为了让所有mapper能够平行进行数据库查询操作，用户需要制定策略，把数据表分成若干段，每个mapper读取一段数据。用户需要考虑的另一个问题是，如何向每个mapper分别传递数据段的参数。

如果要读取的数据段比较大，Hadoop框架很可能会终止长时间运行的任务，除非用户明确向Hadoop框架报告任务进度。

对这样一个概念很简单的任务来讲，需要做的工作却很多。难道没有一个现成的工具可以完成这样的工作吗？实际上，确实存在这样一个工具，那就是Sqoop，本章剩余部分将介绍它的应用。

9.6.3 更好的方法：使用Sqoop

Sqoop是由Cloudera（http://www.cloudera.com）创建的，这家公司提供了许多Hadoop相关的服务，同时也制作自己的Hadoop软件安装包。第11章将会讲到这些内容。

除了提供打包的Hadoop产品外，Cloudera公司也创建了许多大众可以使用的工具，Sqoop便是其中之一。Sqoop的功能正是我们想要实现的，它可以在Hadoop和关系数据库之间拷贝数据。尽管Sqoop最初由Cloudera开发，现在它已经被捐给了Apache软件基金会，其主页地址是http://sqoop.apache.org。

9.7 实践环节：下载并配置 Sqoop

本节将要完成 Sqoop 的安装和配置工作。

(1) 访问 Sqoop 主页，根据读者使用的 Hadoop 版本选取 1.4.1 之后的最稳定版本。下载安装文件。

(2) 把下载到的文件拷贝至目标位置，然后解压缩。

```
$mv sqoop-1.4.1-incubating__hadoop-1.0.0.tar.gz_ /usr/local
$ cd /usr/local
$ tar -xzf sqoop-1.4.1-incubating__hadoop-1.0.0.tar.gz_
```

(3) 制作符号链接。

```
$ ln -s sqoop-1.4.1-incubating__hadoop-1.0.0 sqoop
```

(4) 更新环境变量。

```
$ export SQOOP_HOME=/usr/local/sqoop
$ export PATH=${SQOOP_HOME}/bin:${PATH}
```

(5) 为 MySQL 数据库下载 JDBC 驱动程序，其地址为 http://dev.mysql.com/downloads/connector/j/5.0.html。

(6) 把下载到的 `mysql-connector-java-5.0.8-bin.jar` 文件拷入 Sqoop 的 `lib` 目录:

```
$ cp mysql-connector-java-5.0.8-bin.jar /opt/sqoop/lib
```

(7) 测试 Sqoop 安装是否成功。

```
$ sqoop help
```

上述命令的执行结果如下所示。

```
usage: sqoop COMMAND [ARGS]
Available commands:
  codegen             Generate code to interact with database
records
...
  version             Display version information

See 'sqoop help COMMAND' for information on a specific command.
```

原理分析

Sqoop的安装过程并不复杂。从Sqoop主页下载到所需版本（注意要选用与Hadoop版本匹配的安装包）的安装包后，我们把它拷贝到目标位置并执行解压缩操作。

和其他工具的设置过程一样，我们还需要设置一个环境变量，并把Sqoop的`bin`目录添加到环境变量中。这样，我们既可以直接在shell中调用Sqoop，也可以在一个配置文件中调用Sqoop。

Sqoop要用到MySQL数据库的JDBC驱动，因此我们下载MySQL连接器并把它拷到Sqoop的`lib`目录中。对最常用的数据库来讲，这些就是Sqoop需要的全部设置。如果读者想使用别的数据库，请查阅Sqoop文档。

在完成上述最小安装后，我们在命令行中运行`sqoop`，以验证其可以正常运行。



> 读者可能会看到Sqoop的警告消息，它提醒读者没有定义`HBASE_HOME`变量。因为本书不讨论HBase的相关内容，所以无需设置这些变量，我们会忽略掉这些警告。

1. Sqoop和Hadoop的版本

上节下载Sqoop安装包时，我们多次强调选用合适的版本，在以前下载其他软件时并没有这么在意其版本。这是因为，1.4.1版之前的Sqoop对Hadoop核心类中的某个方法存在依赖关系，而该方法只存在于Cloudera提供的Hadoop安装包或0.21版之后的Hadoop安装包中。

遗憾的是，Hadoop 1.0实际上是0.20分支的延续，这就意味着，Sqoop 1.3可以与Hadoop 0.21配合使用，却无法与0.20或1.0版的Hadoop配合使用。为了避免这些软件版本带来的麻烦，我们推荐读者使用1.4.1之后（包含1.4.1）的Sqoop，它们不会对Hadoop产生依赖关系。

无需再对MySQL进行设置。我们会发现，如果服务器拒绝远程客户端的连接，可以通过Sqoop修改该设置。

2. Sqoop和HDFS

我们可以执行的最简单的导入任务就是把数据表的内容导出到HDFS的结构化文件中。让我们一起来做吧。

9.8　实践环节：把 MySQL 的数据导入 HDFS

本节将演示一个较为简单的例子，从 MySQL 数据表读取数据并将其写入位于 HDFS 上的文件。

(1) 运行 Sqoop，从 MySQL 导出数据，保存在 HDFS 上。

```
$ sqoop import --connect jdbc:mysql://10.0.0.100/hadooptest
--username hadoopuser \ > --password password --table employees
```

9

```
File  Edit  View  Terminal  Help
Warning: $HADOOP_HOME is deprecated.

13/01/05 20:51:47 WARN tool.BaseSqoopTool: Setting your password on the command-line is insecure. Consider using -P instead.
13/01/05 20:51:47 INFO manager.MySQLManager: Preparing to use a MySQL streaming resultset.
13/01/05 20:51:47 INFO tool.CodeGenTool: Beginning code generation
13/01/05 20:51:48 INFO manager.SqlManager: Executing SQL statement: SELECT t.* FROM `employees` AS t LIMIT 1
13/01/05 20:51:48 INFO manager.SqlManager: Executing SQL statement: SELECT t.* FROM `employees` AS t LIMIT 1
13/01/05 20:51:48 INFO orm.CompilationManager: HADOOP_HOME is /opt/hadoop-1.0.3/libexec/..
Note: /tmp/sqoop-hadoop/compile/5104fd58c75a1fcb5833490c2f29707c/employees.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
13/01/05 20:51:48 INFO orm.CompilationManager: Writing jar file: /tmp/sqoop-hadoop/compile/5104fd58c75a1fcb5833490c2f29707c/employees.jar
13/01/05 20:51:48 WARN manager.MySQLManager: It looks like you are importing from mysql.
13/01/05 20:51:48 WARN manager.MySQLManager: This transfer can be faster! Use the --direct
13/01/05 20:51:48 WARN manager.MySQLManager: option to exercise a MySQL-specific fast path.
13/01/05 20:51:48 INFO manager.MySQLManager: Setting zero DATETIME behavior to convertToNull (mysql)
13/01/05 20:51:48 INFO mapreduce.ImportJobBase: Beginning import of employees
13/01/05 20:51:49 INFO db.DataDrivenDBInputFormat: BoundingValsQuery: SELECT MIN(`first_name`), MAX(`first_name`) FROM `employees`
13/01/05 20:51:49 WARN db.TextSplitter: Generating splits for a textual index column.
13/01/05 20:51:49 WARN db.TextSplitter: If your database sorts in a case-insensitive order, this may result in a partial import or duplicate records.
13/01/05 20:51:49 WARN db.TextSplitter: You are strongly encouraged to choose an integral split column.
13/01/05 20:51:49 INFO mapred.JobClient: Running job: job_201301051807_0003
13/01/05 20:51:50 INFO mapred.JobClient:  map 0% reduce 0%
13/01/05 20:52:04 INFO mapred.JobClient:  map 25% reduce 0%
13/01/05 20:52:07 INFO mapred.JobClient:  map 50% reduce 0%
13/01/05 20:52:10 INFO mapred.JobClient:  map 75% reduce 0%
13/01/05 20:52:13 INFO mapred.JobClient:  map 100% reduce 0%
13/01/05 20:52:18 INFO mapred.JobClient: Job complete: job_201301051807_0003
13/01/05 20:52:18 INFO mapred.JobClient: Counters: 18
13/01/05 20:52:18 INFO mapred.JobClient:   Job Counters
13/01/05 20:52:18 INFO mapred.JobClient:     SLOTS_MILLIS_MAPS=23911
13/01/05 20:52:18 INFO mapred.JobClient:     Total time spent by all reduces waiting after reserving slots (ms)=0
13/01/05 20:52:18 INFO mapred.JobClient:     Total time spent by all maps waiting after reserving slots (ms)=0
13/01/05 20:52:18 INFO mapred.JobClient:     Launched map tasks=4
13/01/05 20:52:18 INFO mapred.JobClient:     SLOTS_MILLIS_REDUCES=0
13/01/05 20:52:18 INFO mapred.JobClient:   File Output Format Counters
13/01/05 20:52:18 INFO mapred.JobClient:     Bytes Written=161
13/01/05 20:52:18 INFO mapred.JobClient:   FileSystemCounters
13/01/05 20:52:18 INFO mapred.JobClient:     HDFS_BYTES_READ=513
13/01/05 20:52:18 INFO mapred.JobClient:     FILE_BYTES_WRITTEN=121852
13/01/05 20:52:18 INFO mapred.JobClient:     HDFS_BYTES_WRITTEN=161
13/01/05 20:52:18 INFO mapred.JobClient:   File Input Format Counters
13/01/05 20:52:18 INFO mapred.JobClient:     Bytes Read=0
13/01/05 20:52:18 INFO mapred.JobClient:   Map-Reduce Framework
13/01/05 20:52:18 INFO mapred.JobClient:     Map input records=5
13/01/05 20:52:18 INFO mapred.JobClient:     Physical memory (bytes) snapshot=158652096
13/01/05 20:52:18 INFO mapred.JobClient:     Spilled Records=0
13/01/05 20:52:18 INFO mapred.JobClient:     CPU time spent (ms)=1040
13/01/05 20:52:18 INFO mapred.JobClient:     Total committed heap usage (bytes)=64225280
13/01/05 20:52:18 INFO mapred.JobClient:     Virtual memory (bytes) snapshot=1377288192
13/01/05 20:52:18 INFO mapred.JobClient:     Map output records=5
13/01/05 20:52:18 INFO mapred.JobClient:     SPLIT_RAW_BYTES=513
13/01/05 20:52:18 INFO mapreduce.ImportJobBase: Transferred 161 bytes in 29.8142 seconds (5.4001 bytes/sec)
13/01/05 20:52:18 INFO mapreduce.ImportJobBase: Retrieved 5 records.
hadoop@vm16:~$
```

(2) 检查输出目录。

```
$ hadoop fs -ls employees
```

上述命令的执行结果如下。

```
Found 6 items
-rw-r--r--    3 hadoop supergroup    0 2012-05-21 04:10 /
user/hadoop/employees/_SUCCESS
drwxr-xr-x    - hadoop supergroup    0 2012-05-21 04:10 /
user/hadoop/employees/_logs
-rw-r--r--    3 … /user/hadoop/employees/part-m-00000
-rw-r--r--    3 … /user/hadoop/employees/part-m-00001
-rw-r--r--    3 … /user/hadoop/employees/part-m-00002
-rw-r--r--    3 … /user/hadoop/employees/part-m-00003
```

(3) 查看其中一个结果文件的内容。

```
$ hadoop fs -cat /user/hadoop/employees/part-m-00001
```

上述命令的执行结果如下所示。

```
Bob,Sales,35000,2011-10-01
Camille,Marketing,40000,2003-04-20
```

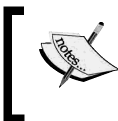
原理分析

我们直截了当的使用一个Sqoop语句完成数据的导入导出。可以看出，Sqoop命令行包含多个选项，本节将逐一解释其用法。

Sqoop的第一个选项是要执行的任务类型，本例中，我们希望把关系数据库存储的数据导入Hadoop。`--connect`选项指定了数据库的JDBC URI，其标准格式为`jdbc:<driver>://<host>/<database>`。很明显，读者需要根据实际情况修改其中的服务器IP或者主机名。

我们使用`--username`和`--password`指定用于连接数据库的用户账号和口令。最后，使用`--table`指定存储目标数据的数据表。这就是需要用户配置的全部内容，剩下的操作由Sqoop完成。

Sqoop的输出相对比较繁琐，但一定要阅读这些内容，因为它很好地揭示了Sqoop的运行过程。



> 重复执行Sqoop可能会引发错误，因为生成的文件已经存在。目前请忽略该错误。

首先，在上述步骤中，我们发现Sqoop提示我们不要使用`--password`选项，因为它本身就是不安全的。Sqoop提供了另一个提示用户输入口令的`-P`命令，今后就用它代替`–password`选项。

Sqoop还抛出一个警告信息，告诉我们使用文本主键是一种糟糕的做法，稍后将详细讨论这个话题。

在完成所有配置之后，Sqoop抛出多个警告信息，但是，我们发现Sqoop成功执行了MapReduce作业。

默认情况下，Sqoop把输出文件放到用户根目录下的某个目录中。存储输出文件的目录名与数据表的名称完全相同。为了验证这一点，我们使用`hadoop fs -ls`检查该目录，结果发现其中包含多个文件。对这么小的数据表而言，输出文件的数量多于我们的预期。请注意，我们稍微紧缩了一下输出，主机就能每行显示一条记录。

接着，我们检查其中一个输出文件的内容，从中发现了输出多个文件的原因。即使数据表很小，Sqoop仍然使用多个mapper读取其内容，因此相应地出现了多个输出文件。默认情况下，Sqoop会使用4个map任务。本例稍微有点特别，通常要导入HDFS的数据非常大。由于我们想把数据拷到HDFS，今后很可能使用MapReduce作业处理这些数据，因此输出多个文件非常合适。

● Mapper及主键

通过手工方式把文本列设为员工数据库的主键列，我们故意设置了这个场景。实际上，一般使用自动增长的、员工的数字ID作为主键。但是，使用文本主键突出了Sqoop处理数据表的方式，以及主键在数据处理过程中的作用。

Sqoop根据主键列的内容决定如何把源数据分段，然后用多个mapper分别读取各段数据。但是，这就意味着，数据分段方法依赖于字符串对比，在大小写不完善的情况下，结果可能有所偏差。最理想的情况是使用数字列作为主键列。

或者，可以使用`-m`选项控制mapper的数量。如果我们使用`-m 1`，Sqoop只会用到一个mapper，这就没必要根据主键列进行数据分段。对于像我们用到这种小数据集，我们可以通过这种方式保证只有一个输出文件。

这不仅仅是一个选项，如果读者不用主键从数据表导出数据，Sqoop将会失败并抛出错误，提醒用户从这种数据表导出数据的唯一办法就是明确设置只使用一个mapper。

● 其他选项

在导入数据时，千万别产生Sqoop无所不能或一无所长的偏见。用户还可以使用其他几个Sqoop选项指定、限定和修改从数据库提取的数据。我们会在后续几节讨论Hive的时候说明这些选项的作用，但请记住，其中大多数选项也可用于把数据库数据导入HDFS。

Sqoop的体系结构

上一节我们已看到Sqoop运作正常，现在有必要花点时间弄清楚它的体系结构及工作原理。在很多方面，Sqoop和Hadoop的交互方式与Hive和Hadoop的交互方式完全相同。它们都是一个独立的客户端程序，可以创建一个或多个MapReduce作业来执行任务。

Sqoop不包含任何服务器进程，我们运行的命令行客户端就是其全部内容。但是，因为它能为手头特定的任务生成合适的MapReduce代码，因此它更能有效利用Hadoop。

上节讲到的基于主键切分RDBMS中源数据表的例子就很好地说明了这一点。Sqoop知道MapReduce作业中将要用到的mapper数量（前几节曾提到，默认设置为4），基于此信息，它可以对源数据表进行智能分块。

假设用4个mapper处理由100万条记录组成的表，那么每个mapper要处理2 50 000条记录。借助主键列的信息，Sqoop可以创建4条SQL语句获取数据，每条语句都限定了目标数据的主键范围。在最简单的情况下，仅需在第一条SQL语句中加入`WHERE id BETWEEN 1 and 250000`这样的声明，并在其他语句中限定不同的`id`范围。

我们还会学习把Hadoop数据导入关系数据库这一反向操作的实现，其中再次用Sqoop实现多个mapper并行获取数据，并优化向关系数据库插入数据的过程。但是，所有这些智能控制都是在运行于Hadoop上的MapReduce作业中实现的。Sqoop命令行客户端程序只负责高效生产MapReduce代码，之后便不参与数据处理过程。

使用Sqoop把数据导入Hive

Sqoop与Hive的集成意义重大，这就可以把关系数据库存储的数据导入新建或现有的Hive数据表。可以通过多种方式实现这一过程，但我们还是从一个简单案例入手。

9.9 实践环节：把 MySQL 数据导出到 Hive

本例中，我们将把一个 MySQL 数据表中的所有数据导入 Hive 中相应的数据表。读者首先需要按照上一章的讲解完成 Hive 的安装和配置工作。

(1) 删除上节创建的输出目录。

```
$ hadoop fs -rmr employees
```

上述命令的执行结果如下。

```
Deleted hdfs://head:9000/user/hadoop/employees
```

(2) 确认 Hive 中不存在 employees 数据表。

```
$ hive -e "show tables like 'employees'"
```

上述命令的执行结果如下。

```
OK
Time taken: 2.318 seconds
```

(3) 使用 Sqoop 执行数据导入任务。

```
$ sqoop import --connect jdbc:mysql://10.0.0.100/hadooptest
--username hadoopuser -P
--table employees --hive-import --hive-table employees
```

```
File Edit View Terminal Help
13/01/05 21:44:42 INFO db.DataDrivenDBInputFormat: BoundingValsQuery: SELECT MIN(`first_name`), MAX(`first_name`) FROM `employees`
13/01/05 21:44:42 WARN db.TextSplitter: Generating splits for a textual index column.
13/01/05 21:44:42 WARN db.TextSplitter: If your database sorts in a case-insensitive order, this may result in a partial import or duplicate records.
13/01/05 21:44:42 WARN db.TextSplitter: You are strongly encouraged to choose an integral split column.
13/01/05 21:44:42 INFO mapred.JobClient: Running job: job_201301052139_0003
13/01/05 21:44:43 INFO mapred.JobClient:  map 0% reduce 0%
13/01/05 21:44:56 INFO mapred.JobClient:  map 25% reduce 0%
13/01/05 21:44:59 INFO mapred.JobClient:  map 50% reduce 0%
13/01/05 21:45:02 INFO mapred.JobClient:  map 75% reduce 0%
13/01/05 21:45:05 INFO mapred.JobClient:  map 100% reduce 0%
13/01/05 21:45:10 INFO mapred.JobClient: Job complete: job_201301052139_0003
13/01/05 21:45:10 INFO mapred.JobClient: Counters: 18
13/01/05 21:45:10 INFO mapred.JobClient:   Job Counters
13/01/05 21:45:10 INFO mapred.JobClient:     SLOTS_MILLIS_MAPS=23343
13/01/05 21:45:10 INFO mapred.JobClient:     Total time spent by all reduces waiting after reserving slots (ms)=0
13/01/05 21:45:10 INFO mapred.JobClient:     Total time spent by all maps waiting after reserving slots (ms)=0
13/01/05 21:45:10 INFO mapred.JobClient:     Launched map tasks=4
13/01/05 21:45:10 INFO mapred.JobClient:     SLOTS_MILLIS_REDUCES=0
13/01/05 21:45:10 INFO mapred.JobClient:   File Output Format Counters
13/01/05 21:45:10 INFO mapred.JobClient:     Bytes Written=161
13/01/05 21:45:10 INFO mapred.JobClient:   FileSystemCounters
13/01/05 21:45:10 INFO mapred.JobClient:     HDFS_BYTES_READ=513
13/01/05 21:45:10 INFO mapred.JobClient:     FILE_BYTES_WRITTEN=121864
13/01/05 21:45:10 INFO mapred.JobClient:     HDFS_BYTES_WRITTEN=161
13/01/05 21:45:10 INFO mapred.JobClient:   File Input Format Counters
13/01/05 21:45:10 INFO mapred.JobClient:     Bytes Read=0
13/01/05 21:45:10 INFO mapred.JobClient:   Map-Reduce Framework
13/01/05 21:45:10 INFO mapred.JobClient:     Map input records=5
13/01/05 21:45:10 INFO mapred.JobClient:     Physical memory (bytes) snapshot=156000256
13/01/05 21:45:10 INFO mapred.JobClient:     Spilled Records=0
13/01/05 21:45:10 INFO mapred.JobClient:     CPU time spent (ms)=1000
13/01/05 21:45:10 INFO mapred.JobClient:     Total committed heap usage (bytes)=64225280
13/01/05 21:45:10 INFO mapred.JobClient:     Virtual memory (bytes) snapshot=1376374784
13/01/05 21:45:10 INFO mapred.JobClient:     Map output records=5
13/01/05 21:45:10 INFO mapred.JobClient:     SPLIT_RAW_BYTES=513
13/01/05 21:45:10 INFO mapreduce.ImportJobBase: Transferred 161 bytes in 28.7967 seconds (5.5929 bytes/sec)
13/01/05 21:45:10 INFO mapreduce.ImportJobBase: Retrieved 5 records.
13/01/05 21:45:10 INFO manager.SqlManager: Executing SQL statement: SELECT t.* FROM `employees` AS t LIMIT 1
13/01/05 21:45:10 WARN hive.TableDefWriter: Column start_date had to be cast to a less precise type in Hive
13/01/05 21:45:10 INFO hive.HiveImport: Removing temporary files from import process: hdfs://localhost:9000/user/hadoop/employees/_logs
13/01/05 21:45:10 INFO hive.HiveImport: Loading uploaded data into Hive
13/01/05 21:45:11 INFO hive.HiveImport: WARNING: org.apache.hadoop.metrics.jvm.EventCounter is deprecated. Please use org.apache.hadoop.log.metrics.EventCoun
ter in all the log4j.properties files.
13/01/05 21:45:12 INFO hive.HiveImport: Logging initialized using configuration in jar:file:/opt/hive-0.8.1/lib/hive-common-0.8.1.jar!/hive-log4j.properties
13/01/05 21:45:12 INFO hive.HiveImport: Hive history file=/tmp/hadoop/hive_job_log_hadoop_201301052145_1438537341.txt
13/01/05 21:45:15 INFO hive.HiveImport: OK
13/01/05 21:45:15 INFO hive.HiveImport: Time taken: 2.842 seconds
13/01/05 21:45:15 INFO hive.HiveImport: Loading data to table default.employees
13/01/05 21:45:15 INFO hive.HiveImport: OK
13/01/05 21:45:15 INFO hive.HiveImport: Time taken: 0.395 seconds
13/01/05 21:45:15 INFO hive.HiveImport: Hive import complete.
13/01/05 21:45:15 INFO hive.HiveImport: Export directory is empty, removing it.
hadoop@vm16:~$
hadoop@vm16:~$
```

(4) 检查 Hive 中的内容。

```
$ hive -e "select * from employees"
```

上述命令的执行结果如下。

```
OK
Alice   Engineering   50000   2009-03-12
Camille   Marketing    40000   2003-04-20
David   Executive   75000   2001-03-20
Erica   Support   34000   2011-07-07
Time taken: 2.739 seconds
```

(5) 检查 Hive 中已创建的数据表。

```
$ hive -e "describe employees"
```

上述命令的执行结果如下。

```
OK
first_name    string
dept    string
salary    int
start_date    string
Time taken: 2.553 seconds
```

原理分析

本例中用到了Sqoop的两个新选项：--hive-import选项指明数据的目标存储位置是Hive而非HDFS，--hive-table to则指明了Hive中用于存储导入数据的数据表。

事实上，如果Hive中存储导入数据的表与--table选项指定的源数据表一致的话，无需专门指明Hive表名。但是，--hive-table to选项使它变得更明确，因此我们通常都要用到该选项。

和以前一样，一定要从头到尾仔细阅读Sqoop的输出，因为它能揭示Sqoop的运行情况，最后几行表明数据成功导入了新建的Hive表。

我们看到，Sqoop从MySQL获取了5行数据，然后就把它们拷贝到HDFS并导入Hive。接下来我们会讨论关于类型转换的警告。

Sqoop完成数据导入任务后，我们从新建的Hive表中读取数据以确认结果和预期一致。接着，我们检查了新建的employees表的定义。

这时，出现了一些奇怪的现象：start_date列在MySQL数据库中是DATE类型，而现在却成了string类型。

Sqoop运行时输出的警告信息可以解释这个现象。

```
12/05/23 13:06:33 WARN hive.TableDefWriter: Column start_date had to be
cast to a less precise type in Hive
```

出现这种现象的原因在于，除TIMESTAMP之外，Hive不支持其他临时数据类型。如果被导入的数据是与时间或日期相关的其他类型，Sqoop会把它转换成string类型。稍后我们会介绍处理这种情况的办法。

本例提到的操作非常常见，但我们并非总是想把整个数据表导入Hive。有时，我们只想把特定几列数据导入Hive，或者在数据导入之前进行筛选以减少数据项。Sqoop可以用于这两种场景。

9.10　实践环节：有选择性的导入数据

接下来，我们将学习如何通过条件语句有选择地把 MySQL 数据导入 Hive。

(1) 删掉 Hadoop 中现有的 employees 目录：

```
$ hadoop fs -rmr employees
```

上述命令的执行结果如下：

```
Deleted hdfs://head:9000/user/hadoop/employees
```

(2) 使用条件语句导入选中的数据列。

```
sqoop import --connect jdbc:mysql://10.0.0.100/hadooptest
--username hadoopuser -P
--table employees --columns first_name,salary
--where "salary > 45000"
--hive-import --hive-table salary
```

上述命令的执行结果如下。

```
12/05/23 15:02:03 INFO hive.HiveImport: Hive import complete.
```

(3) 检查新建的 Hive 数据表。

```
$ hive -e "describe salary"
```

上述命令的执行结果如下。

```
OK
first_name    string
salary    int
Time taken: 2.57 seconds
```

(4) 检查 Hive 表中的导入数据。

```
$ hive -e "select * from salary"
```

上述命令的执行结果如下。

```
OK
Alice    50000
David    75000
Time taken: 2.754 seconds
```

原理分析

这次，我们在Sqoop命令中加入`--columns`选项，指定将哪几列的数据导入Hive。该参数的值是以逗号为分隔符的列表。

我们还用到了`--where`选项，用它指定数据筛选条件，其作用相当于SQL语句中的`WHERE`子句。

组合使用上述两个选项，意味着Sqoop只会把薪水值大于`WHERE`子句指定的门限值的员工姓名和薪水导入Hive。

在成功执行Sqoop命令后，我们检查了Hive中创建的数据表。我们发现，表中确实只包含姓名和薪水两列，接着我们输出表的内容，以验证数据导入过程正确地应用了判断语句。

数据类型的问题

我们在第8章曾提到，Hive仅支持部分常用的SQL数据类型。尤其是，目前Hive尚不支持DATE和DATETIME数据类型，尽管这确实是Hive存在的一个问题。因此，今后很有可能Hive会加入对这两个数据类型的支持。本章前几节讲到的例子就受到了这个因素的影响。尽管start_date列在MySQL中的数据类型为DATE，Sqoop在导入数据时抛出类型转换的警告，导致该列在Hive中的数据类型变为STRING。

Sqoop提供了一个有用的选项，也就是说，我们可以使用--map-column-hive选项明确指定Hive表中新建数据列的数据类型。

9.11 实践环节：使用数据类型映射

接下来，我们使用类型映射改进数据导入过程。

(1) 删除 Hadoop 上已有的 employees 目录。

```
$ hadoop fs -rmr employees
```

(2) 明确使用类型映射执行 Sqoop 命令。

```
sqoop import --connect jdbc:mysql://10.0.0.100/hadooptest
--username hadoopuser
-P --table employees
--hive-import --hive-table employees
--map-column-hive start_date=timestamp
```

上述命令的执行结果如下。

```
12/05/23 14:53:38 INFO hive.HiveImport: Hive import complete.
```

(3) 检查新建数据表的定义。

```
$ hive -e "describe employees"
```

上述命令的执行结果如下。

```
OK
first_name    string
dept    string
salary    int
start_date    timestamp
Time taken: 2.547 seconds
```

(4) 检查导入到 Hive 表中的数据。

```
$ hive -e "select * from employees";
```

上述命令的执行结果如下。

```
OK
Failed with exception java.io.IOException:java.lang.
IllegalArgumentException: Timestamp format must be yyyy-mm-dd
hh:mm:ss[.fffffffff]
Time taken: 2.73 seconds
```

原理分析

本节用到的Sqoop命令行与最初从MySQL向Hive导入数据的命令类似，只是新增了一个列映射声明。我们指定start_date列的数据类型为TIMESTAMP，除此之外，还可以增加其他声明。该选项的值是以逗号为分隔符的列表。

在确认Sqoop命令执行成功后，我们检查了新建的Hive数据表并验证类型映射确实发挥了作用，start_date列的数据类型确实是TIMESTAMP。

接着，我们尝试从Hive表中读取数据，该过程以失败而告终，Hive抛出一个类型格式不匹配的错误信息。

仔细想想，这也没什么值得惊讶的。尽管我们指定目标列的数据类型为TIMESTAMP，但从MySQL导入的数据类型实为DATE，其中并不包含TIMESTAMP类型所需的时间元素。这个教训要牢记。保证使用正确的数据类型映射只是解决数据类型问题的一个方面，还必须同时确保目标数据符合指定的数据类型的要求。

9.12 实践环节：通过原始查询导入数据

本节将学习如何使用原始的 SQL 查询筛选目标数据，并将其导入 Hive 表。

(1) 删除 Hadoop 上现有的 employees 目录。

```
$ hadoop fs -rmr employees
```

(2) 删除 Hive 中已有的 employees 数据表。

```
$ hive -e 'drop table employees'
```

(3) 使用原始查询选定目标数据，使用 Sqoop 命令将其导入 Hive。

```
sqoop import --connect jdbc:mysql://10.0.0.100/hadooptest
--username hadoopuser -P
--target-dir employees
```

```
--query 'select first_name, dept, salary,
timestamp(start_date) as start_date from employees where
$CONDITIONS'
--hive-import --hive-table employees
--map-column-hive start_date=timestamp -m 1
```

(4) 检查 Hive 中刚创建的 employees 数据表。

```
$ hive -e "describe employees"
```

上述命令的执行结果如下。

```
OK
first_name    string
dept    string
salary    int
start_date    timestamp
Time taken: 2.591 seconds
```

(5) 检查 employees 数据表中的数据。

```
$ hive -e "select * from employees"
```

上述命令的执行结果如下。

```
OK
Alice    Engineering    50000    2009-03-12 00:00:00
Bob    Sales    35000    2011-10-01 00:00:00
Camille    Marketing    40000    2003-04-20 00:00:00
David    Executive    75000    2001-03-20 00:00:00
Erica    Support    34000    2011-07-07 00:00:00
Time taken: 2.709 seconds
```

原理分析

为了达到目标，我们使用了一种截然不同的方法完成数据导入任务。以前，我们指定目标数据表，然后使用Sqoop命令导入其部分或全部数据。与此不同，本例使用`--query`选项明确定义一个SQL查询语句，通过该语句指明要导入的数据范围。

在SQL语句中，我们选中了源数据表的所有列，并使用`timestamp()`方法将`start_date`列的数据转换为合适的类型（请注意，该方法仅在日期的基础上加入`00:00`时间元素）。我们为该函数的执行结果取了个别名，便于在类型映射选项中引用这些数据。

因为我们没有指定`--table`选项，不得不加入`--target-dir`选项以指定HDFS上的输出目录。

Sqoop要求我们必须在SQL语句中加入WHERE子句，即使根本不会用到这个条件语句。不指定--table选项不仅意味着Sqoop无法自动生成存储导出数据的路径名，而且Sqoop不知道从哪个表中获取数据，进而也不知道如何为多个mapper切分数据。在--where选项后紧接$CONDITIONS变量的意义在于，$CONDITIONS变量会为Sqoop提供切分数据表所需的信息。

本例中，我们采用了不同的处理方式，明确地将mapper的数量设为1，避免了使用数据分块子句。

在Sqoop执行完成之后，我们检查了Hive数据表的定义，结果表明，所有列的数据类型都完全正确。接着，我们又检查了表中数据，这次查询终于成功了，因为start_date列的值已转换成了TIMESTAMP类型。



我们曾提到，用户可以使用Sqoop只提取数据库中的部分数据，Sqoop提供的query、where和columns选项可以限定待提取数据的范围。需要注意的是，这些选项可以用于所有Sqoop数据导入任务，而与导入数据的目标存储位置无关。

一展身手

尽管示例中的数据集规模太小，完全不需要对其进行数据切分，但$CONDITIONS变量仍是个很重要的工具。请修改上例，在Sqoop语句中使用多个mapper，并明确定义数据分块语句。

1. Sqoop和Hive数据分块

我们在第8章对Hive数据分块进行了大量讨论，并强调了它对优化大规模数据表的查询效率具有重要作用。Sqoop支持在Hive中进行数据分块，这是个令人振奋的好消息，但是不要高兴得太早，它对Hive数据分块的支持并不完整。

为了把关系数据库的数据导入Hive分区表，我们使用--hive- partition-key选项指定分区列，使用--hive-partition- value选项指定分区值，Sqoop命令会根据该值决定把导入数据插入哪个分区表中。

这是一个好办法，但用户必须提供相应的Sqoop语句才能把导入数据插入某个分区表。目前Sqoop尚不提供对Hive自动分区的支持。如果读者想把某个数据集插入多个Hive分区表，只能多次通过Sqoop语句逐表插入。

2. 字段分隔符和文本行分隔符

截至目前，我们一直暗中依赖于某些默认设置，但现在应该探讨一下这些内容。原始文本文

件以制表符为分隔符，但读者可能注意到，导入HDFS的数据却以逗号为分隔符。如果读者查看/user/hive/ warehouse/employees（这是Hive在HDFS上存储源文件的默认位置）目录中的文件，文件中的记录又以ASCII码001作为分隔符。这到底是怎么回事呢？

在第一个例子中，我们使用Sqoop的默认分隔符，也就是说，使用逗号作为字段间的分隔符，并使用\n作为记录间的分隔符。但是，当Sqoop把数据导入Hive之后，却又使用Hive的默认值，也就是使用ASCII码001（^A）来分隔字段。

我们可以使用下列Sqoop选项明确设置分隔符。

- fields-terminated-by：设置字段间分隔符。
- lines-terminated-by：设置文本行分隔符。
- escaped-by：指定转义字符（例如，\）。
- enclosed-by：用于封闭字段的字符（例如，"）。
- optionally-enclosed-by：与上个选项相似，但不是强制使用的。
- mysql-delimiters: 使用MySQL默认值的快捷设置。

这些分隔符看似多得吓人，其实它们并不像这些术语那样难懂，有SQL使用经验的人应该对这些概念和用法非常熟悉。前几个选项无需过多解释，但封闭字符和可选封闭字符的含义却不是那么明显。

这两个选项用于给定字段中包含特殊字符的情况。例如，某个文件以逗号为分隔符，其中某列的数据类型为string，而该列的某个值中包含有逗号。在这种情况下，我们要把该字符串放入引号，这样才能保证使用逗号作为字段分隔符。如果所有字段都需要使用封闭字符，那么就要使用“enclosed-by”选项，而如果只是部分字段要用到封闭字符，那么就应该使用“optionally-enclosed-by”选项。

9.13　从 Hadoop 导出数据

我们曾提到，Hadoop和关系数据库之间的数据流是一个线性的单向过程，这是非常罕见的。实际上，我们更常遇到把Hadoop处理过的数据插入关系数据库的情况。接下来，我们将讨论这个问题。

9.13.1　在reducer中把数据写入关系数据库

考虑一下如何把MapReduce作业的输出数据拷入关系数据库，我们发现，其解决方法和向Hadoop导入数据的方法类似。

一种直观的方法是修改reducer代码，使其生成每个键及对应的值之后，直接通过JDBC把它们插入数据库。我们不必像从关系数据库导入数据那样考虑如何对源数据分段，但仍要考虑数据

库能够承载的负荷，以及是否需要考虑长时间运行任务的超时问题。此外，与mapper遇到的问题一样，这种方法要对数据库执行多次查询，而每次查询只插入一条数据，其效率远低于成批操作。

9.13.2　利用reducer输出SQL数据文件

通常，较好的解决办法不会围绕负责生成输出文件的MapReduce作业下功夫，而重点在于如何利用它。

所有关系数据库都可通过定制工具或是`LOAD DATA`语句接收源文件里的数据。因此，我们可以改写reducer的数据输出方法，使其更易于被插入关系数据库。这就避免了考虑reducer带给数据库的负担，以及如何处理长时间运行任务的麻烦，但需要在MapReduce作业完成之后添加一个步骤，把输出文件的数据导入关系数据库。

9.13.3　仍是最好的方法

Sqoop还可以用于从Hadoop向关系数据库导入数据，这应该没什么奇怪的。如果读者曾浏览过Sqoop使用帮助或其在线文档，就会觉得这是理所当然的。

9.14　实践环节：把 Hadoop 数据导入 MySQL

本节将演示如何把 HDFS 文件数据导入 MySQL 数据表。

(1) 创建 `newemployees.tsv` 文件，其以制表符为分隔符，内容如下所示。

```
Frances    Operations    34000    2012-03-01
Greg    Engineering    60000    2003-11-18
Harry    Intern    22000    2012-05-15
Iris    Executive    80000    2001-04-08
Jan    Support    28500    2009-03-30
```

(2) 在 HDFS 上新建一个目录，将 `newemployees.tsv` 拷入该目录。

```
$hadoop fs -mkdir edata
$ hadoop fs -put newemployees.tsv edata/newemployees.tsv
```

(3) 确认 employees 数据表中的记录数。

```
$ echo "select count(*) from employees" |
mysql -u hadoopuser -p hadooptest
```

上述命令的执行结果如下。

```
Enter password:
count(*)
5
```

(4) 运行 Sqoop，将 HDFS 上的文件数据导入 MySQL。

```
$ sqoop export --connect jdbc:mysql://10.0.0.100/hadooptest
--username hadoopuser    -P --table employees
--export-dir edata --input-fields-terminated-by '\t'
```

上述命令的执行结果如下。

```
12/05/27 07:52:22 INFO mapreduce.ExportJobBase: Exported 5
records.
```

(5) 再次检查 employees 数据表中的记录数。

```
Echo "select count(*) from employees"
| mysql -u hadoopuser -p hadooptest
```

上述命令的执行结果如下。

```
Enter password:
count(*)
10
```

(6) 检查 employees 数据表中的数据内容。

```
$ echo "select * from employees"
| mysql -u hadoopuser -p hadooptest
```

上述命令的执行结果如下。

```
Enter password:
first_name    dept    salary    start_date
Alice    Engineering    50000    2009-03-12
...
Frances    Operations    34000    2012-03-01
Greg    Engineering    60000    2003-11-18
Harry    Intern    22000    2012-05-15
Iris    Executive    80000    2001-04-08
Jan    Support    28500    2009-03-30
```


原理分析

我们首先创建一个数据文件，并在该文件中加入5条新员工数据。在HDFS上创建一个存储该数据文件的目录。

在运行导出任务之前，我们确认MySQL数据表中只存储了原来的5个员工数据。

Sqoop命令的结构与之前类似，最大的变化在于，使用的是`export`命令。顾名思义，导出

Hadoop数据并插入关系数据库。

本例用到了几个选项，它们和之前的用法相同，主要用于设置要连接的数据库地址，连接数据库使用的用户帐号和口令，以及要把数据插入哪个数据表。

因为我们要从HDFS导出数据，因此需要通过`--export-dir`选项指定包含待导出数据文件的目录。Sqoop会生成一个MapReduce作业，然后把该目录下的所有文件全部导出，因此导出目录下有一个文件还是多个文件对于导出任务没有多大影响。默认情况下，Sqoop会使用4个mapper，如果读者需要导出大量文件，那么增大mapper的数量会提高作业的运行效率。读者可以尝试一下，但要确保数据库能够承载这么多的并发连接。

Sqoop用到的最后一个选项指明了源文件使用的分隔符，本例使用制表符作为分隔符。读者需要自己指明数据文件的格式，Sqoop会认为每条记录的字段数和数据表的列数保持一致（尽管Sqoop支持空字段值），并且各字段由指定的分隔符分开。

在Sqoop命令成功执行后，它向用户报告已导出5条记录。随后，我们使用`mysql`工具检查数据表中的当前记录总数，并接着查看数据表的内容，以确认新员工数据和原有的员工数据都存在于数据表中。

1. Sqoop导入和导出的区别

尽管Sqoop导入和导出在概念上和命令行调用方面非常相似，但它们之间存在许多重要区别，值得我们深入研究。

首先，在使用Sqoop导入数据时，Sqoop对数据结构和数据类型的了解更详细一些。但是，在导出数据时，Sqoop只知道源文件位置以及字段和记录间的分隔符。此外，Sqoop导入数据时可根据源数据表的名称和结构自动新建一个Hive数据表，但只能把导出数据插入关系数据库中的已有数据表。

使用Sqoop导入数据时，Sqoop可以意识到源数据与Hive表中各列的数据类型是否匹配。尽管之前对DATE类型和TIMESTAMP类型的介绍表明Sqoop在这方面做得不是很完美，但至少不必等到数据已插入Hive表之后才发现问题。而对Sqoop导出而言，它只负责高效读取各字段内容，无需理解数据类型。假如用户很幸运，要导出的数据格式非常规整，可能不会有什么麻烦。但大多数人不得不考虑导出数据的格式转换，尤其是数据中存在空字段和默认值的情况下。Sqoop文档深度讲解了这些内容，读者有必要认真阅读。

2. 插入和更新的对比

上个例子比较简单，我们向关系数据库导入的是全新数据集，它可以与表中原有数据同时存在。默认情况下，Sqoop导出数据时进行了一些列扩展，它把每条记录当做一行新数据加入数据表。

但是，如果我们以后想更新数据时，例如，年底的时候为员工涨了工资，此时该怎么办呢？由于我们把`first_name`定义为数据表的主键，如果要插入数据的姓名字段与现有员工姓名相同，那么插入操作就会失败。

在这种情况下，我们可以在Sqoop命令中设置`--update-key`选项，通过该选项指定主键，Sqoop就会根据该键（也可以是以逗号分隔的多个键）生成`UPDATE`语句，达到更新原有记录部分字段的目的。



在这种模式下，Sqoop将会略去与现有键的值不匹配的记录，而且如果被更新的数据多于一行，Sqoop也不会抛出错误信息。

假如读者想在更新现有数据的同时，在数据表中加入并不存在的新记录，可以将`--update-mode`选项设置成`allowinsert`。

一展身手

新建一个数据文件，其中包含3条新员工记录，同时对2位老员工的薪水进行了调整。使用Sqoop的导入模式将新员工数据加入数据表，并更新老员工的薪水数据。

3. Sqoop和Hive导出

从上例可以看出，Sqoop目前无法直接将Hive数据表导入关系数据库，读者对此不会感到特别吃惊。更确切地说，Sqoop中提供了`--hive-import`这样的导入选项，却没有提供类似的导出选项。

但是，在某些情况下，我们可以解决这个问题。假如Hive数据表以文本形式存储数据，我们可以设置Sqoop，使其指向HDFS上存储这些表数据文件的位置。如果数据表存储的是外部数据，这就更简单了。但是如果Hive数据表进行了复杂的分区操作，那么文件目录结构就更为复杂。

Hive数据表中也可以存储二进制SequenceFile，Sqoop的一个缺点在于其目前无法透明导出这种格式的数据。

9.15 实践环节：把 Hive 数据导入 MySQL

先别管 Sqoop 的这些缺点，本节将展示如何使用 Sqoop 直接导出 Hive 数据表的数据。

(1) 删除 employees 数据表中的所有数据。

```
$ echo "truncate employees" | mysql -u hadoopuser -p hadooptest
```

上述命令的执行结果如下。

```
Query OK, 0 rows affected (0.01 sec)
```

(2) 查看 hive/warehouse/employees 的内容。

```
$ hadoop fs -ls /user/hive/warehouse/employees
```

上述命令的执行结果如下。

```
Found 1 items
... /user/hive/warehouse/employees/part-m-00000
```

(3) 使用 Sqoop 命令执行数据导出任务。

```
sqoop export --connect jdbc:mysql://10.0.0.100/hadooptest
--username hadoopuser -P --table employees \
--export-dir /user/hive/warehouse/employees
--input-fields-terminated-by '\001'
--input-lines-terminated-by '\n'
```

原理分析

首先，我们删除MySQL数据库中employees数据表的所有数据，然后确认该数据表位于/user/hive/warehouse/employees目录。



请注意，Sqoop可能会在该路径下创建一个空白文件，其文件后缀为_SUCCESS。如果该目录下确实存在这个文件的话，一定要在运行Sqoop之前把它删除。

Sqoop的export命令与以前的用法相似，唯一的区别在于，本例中源数据文件的存储位置发生了变化，并且明确指定了字段和文本行分隔符。回忆一下，在默认情况下，Hive分别使用ASCII码001和\n作为字段分隔符和文本行分隔符。同时，由于我们向Hive导入数据时使用了其他分隔符，因此需要对其进行检查。

运行Sqoop命令，发现在创建java.sql.Date实例时出错，错误原因是IllegalArgument-Exceptions。

这个问题与之前遇到的问题刚好相反：原来把MySQL数据导入Hive数据表时，由于Hive不支持MySQL中的DATE类型，我们将其转换成Hive中可用的TIMESTAMP类型。在把Hive数据重新导入MySQL数据表时，我们却遇到了把TIMESTAMP类型的数据插入DATE数据列的问题。在不进行数据转换的情况下，这是无法实现的。

这两个例子给我们的启发就是，单向数据转换只对单向数据流有效。一旦需要进行双向数据传输，Hive和关系数据库之间的数据类型不匹配问题会带来很多麻烦，必须在数据传输之前首先进行数据类型转换。

9.16　实践环节：改进 mapper 并重新运行数据导出命令

但是，这次我们不再进行数据类型转换，而要做一些更有意义的事——使用 Hive 和 MySQL 都支持的数据类型修改 employees 数据表的定义。

(1) 启动 mysql 命令行程序。

```
$ mysql -u hadoopuser -p hadooptest
Enter password:
```

(2) 修改 start_date 列的数据类型：

```
mysql> alter table employees modify column start_date timestamp;
```

上述命令的执行结果如下。

```
Query OK, 0 rows affected (0.02 sec)
Records: 0    Duplicates: 0    Warnings: 0
```

(3) 输出 employees 数据表的定义。

```
mysql> describe employees;
```

9

(4) 退出 mysql 程序。

```
mysql> quit;
```

(5) 执行 Sqoop 导出命令。

```
sqoop export --connect jdbc:mysql://10.0.0.100/hadooptest
--username hadoopuser -P -table employees
--export-dir /user/hive/warehouse/employees
--input-fields-terminated-by '\001'
--input-lines-terminated-by '\n'
```

上述命令的执行结果如下。

```
12/05/27 09:17:39 INFO mapreduce.ExportJobBase: Exported 10
records.
```

(6) 检查 MySQL 数据库中的记录总数。

```
$ echo "select count(*) from employees"
| mysql -u hadoopuser -p hadooptest
```

上述命令的执行结果如下所示。

```
Enter password:
count(*)
10
```

原理分析

在最后一次执行Sqoop导出命令之前，我们使用mysql命令行工具连接数据库，并修改了start_date列的数据类型。当然，千万不能随便在生产系统中执行这样的修改，但由于我们使用的空白的测试数据表，因此不会带来什么问题。

在完成这些修改之后，我们重新运行Sqoop导出任务，这一次终于成功了。

Sqoop的其他功能

Sqoop还有许多其他功能。虽然我们不再详细讨论这些内容，但仍要强调一下它们，以便感兴趣的读者在Sqoop文档中查阅相关内容。

- 增量合并

一直以来，我们接触的示例都是比较极端的。确实，我们遇到的大多数情况都是向空白数据表导入数据。目前也介绍了一些处理增量数据的方法，但Sqoop提供了其他针对持续的数据导入任务的支持。

Sqoop提出了增量导入的概念。举个例子来说明它的含义，如果数据导入任务与日期有关，只有某天之后的记录才会被导入。这就需要使用包括Sqoop在内的工具构建一个长期运行的工作流。

- 避免部分导出

我们已经发现，把Hadoop数据导出到关系数据库时有可能会出现错误。在我们的例子中，这个问题没什么大不了的，因为该错误导致所有记录都无法导出。但往往会出现导出部分数据之后，导出任务却中途出现故障的情况，这时数据库中只插入了部分数据。

Sqoop使用分段表来消除这个风险，它把所有数据导入这个二级表，只有成功插入所有数据后，才会使用一次事务把分段表的数据全部移到主表中。该方案对可靠性较差的工作负载非常有效，却也带来了一些重要的限制，例如它不支持更新模式。对数据量很大的导入任务，由于使用了一个长时间运行的事务，也会对关系数据库的性能和负荷带来影响。

- 使用Sqoop作为代码生成器

我们一直都忽视了Sqoop运行过程中出现的一个错误，刚才还偶然遇到了这类错误——由于Sqoop所需代码早已存在而抛出的异常。

在执行数据导入任务时，Sqoop会生成Jave类文件，通过这些Java代码来访问文件中的字段和记录。Sqoop在内部使用这些类，但这些代码并非只能用于Sqoop调用。事实上，通过Sqoop的`codegen`命令，可以在数据导出任务结束之后重新生成任务用到的Java类。

9.17　在 AWS 上使用 Sqoop

截至目前，本章还未提及AWS，这是因为Sqoop在AWS上的运行情况并无特别之处。我们可以在EC2主机上运行Sqoop，就像是在本地主机上运行Sqoop一样。而且，运行在EC2主机上的Sqoop既可以访问手工创建的Hadoop集群，也可以访问EMR托管的Hadoop集群。如果需要的话，还可以在这些Hadoop集群上运行Hive。唯一需要考虑的问题是安全组访问策略（security group

access），因为许多默认的EC2配置选项拒绝多数关系数据库使用的端口（MySQL的默认端口是3306）发起的连接。但是，与Hadoop集群和MySQL数据库分别位于防火墙或其他网络安全边界的两端相比，这根本就不是什么问题。

使用RDS

以前，我们从没提过AWS的**RDS**（Relational Database Service，关系数据库服务），但现在很有必要介绍一下它。RDS提供了托管在云端的关系数据库，用户可以根据需要选用MySQL、Oracle和Microsoft SQL Server。使用RDS服务，用户不必再为数据库的安装、配置、管理而耗费精力，而且可以通过RDS控制台或命令行工具启动数据库实例。用户执行使用数据库客户端打开数据库即可创建数据表，并操作数据。

组合使用RDS和EMR更能充分发挥它们的强大作用，这些托管服务省去了手动管理服务器集群的麻烦。如果读者要用到关系数据库，而又不想在数据库管理方面浪费精力，可以尝试使用RDS。

如果读者使用EC2主机生成数据，或在S3中存储数据，那么组合使用RDS和EMR的效果更为明显。Amazon公司有一个通用规则，在相同主机上，用户可以免费在不同服务之间传输数据。因此，用户可以使用一批EC2主机生成大量数据，然后把它们插入RDS关系数据库以便查询，并把它们存储在EMR作为档案进行长期分析。把数据导入存储系统和处理系统通常很有挑战性，经常会付出极大的代价，尤其是需要在不同商用系统中移动数据的时候。使用EC2、RDS和EMR相互配合，可以减低这些成本。

9.18　小结

本章介绍了Hadoop和关系数据库的集成使用。特别是，我们研究了最常用的案例，并发现Hadoop和关系数据库都是广受赞誉的技术。我们想了几种从关系数据库向HDFS导出数据的办法，并意识到文本主键无法实现数据分段，并且需要特别考虑如何处理长时间运行的任务。

接着，我们介绍了Sqoop。这是Cloudera公司捐献给Apache软件基金会的一个工具，它提供了实现上述数据移动的框架。使用Sqoop可以实现从MySQL向HDFS或Hive的数据导入，但我们必须考虑数据类型是否兼容的问题。Sqoop还可以实现相反操作，也就是把数据从HDFS导入MySQL数据库。该过程要考虑更多问题，简单来讲，包括文件格式的问题，插入数据与更新数据的区别，以及Sqoop兼容性的问题。此外，还介绍了Sqoop的代码生成和增量合并功能。

关系数据库在大多数IT基础设施中处于重要地位，甚至是十分关键的。但是，并不等于系统中的其他组件并不重要。最近，有个问题变得日益突出，那就是网页服务器和其他应用程序生成的日志文件越来越多。下一章我们将介绍Hadoop如何存储并处理这些日志数据。

第 10 章

使用Flume收集数据

前两章，我们介绍了Hive和Sqoop为Hadoop实现的类似关系数据库的接口，使用它们可以与“真正的”数据库交换数据。尽管这是非常常见的情况，但我们一定还会遇到想把许多不同类别的源数据导入Hadoop的情况。

本章包括以下内容：

- Hadoop通常处理的数据类别；
- 把数据导入Hadoop的简单方法；
- Apache Flume为何会简化数据导入任务；
- 由简到繁的Flume配置；
- 需要考虑的非技术问题，如数据的生命周期。

10.1 关于 AWS 的说明

全书中，本章涉及AWS的内容最少。实际上，除本节之外，本章基本不涉及AWS的内容。Amazon没有提供类似Flume的服务，因此不存在专为AWS实现的类似产品可供研究。另一方面，无论在本地主机或是EC2虚拟机上运行Flume，其工作原理完全相同。因此，本章其余部分不再对示例的运行环境提任何要求，它们在任何环境中的运行完全相同。

10.2 无处不在的数据

在讨论Hadoop与其他系统的集成问题的时候，很容易把它想成一对一的模式。数据从一个系统中流出，经过Hadoop系统的处理，然后被传给另一个系统。

最初的时候，事情可能确实如此，但实际情况往往是，要用一系列相互配合的部件循环往复

地处理数据。本章关注的核心问题是如何构建这种可维护的复杂网络。

10.2.1　数据类别

为了便于讨论，我们把数据分成以下两大类。

- **网络流量**：数据由某个系统生成并通过网络连接进行传输。
- **文件数据**：数据由某个系统生成并被写入文件系统上的文件。

我们认为，这两类数据的区别仅仅在于数据获取方式不同。

10.2.2　把网络流量导入Hadoop

我们讲的网络数据，指的是通过HTTP连接从网络服务器获取的信息，通过客户端应用程序获取的数据库内容，或者通过数据总线发送的消息。在以上各种情况中，要么是通过客户端应用程序从网络上获取数据，要么通过监听某个端口等待数据。



接下来的几个例子中，我们会使用curl程序接收或发送网络数据。确保主机上已安装了该软件，要是还没安装的话，请提前安装。

10.3　实践环节：把网络服务器数据导入 Hadoop

接下来，我们将介绍如何简单地把网络服务器数据拷贝至Hadoop。

(1) 从NameNode的网页接口获取文本数据，并把它保存到一个本地文件。

```
$ curl localhost:50070 > web.txt
```

(2) 查看文件大小。

```
$ ls -ldh web.txt
```

上述命令的执行结果如下。

```
-rw-r--r-- 1 hadoop hadoop 246 Aug 19 08:53 web.txt
```

(3) 把该文件拷贝到HDFS。

```
$ hadoop fs -put web.txt web.txt
```

(4) 查看HDFS上的文件副本。

```
$ hadoop fs -ls
```

上述命令的执行结果如下。

```
Found 1 items
-rw-r--r--    1 hadoop supergroup    246 2012-08-19 08:53 /
user/hadoop/web.txt
```

原理分析

上例并没讲到什么新奇的内容。我们使用curl程序从内置的网页服务器获取NameNode网页接口的页面内容，并把它保存为一个本地文件。我们检查了文件大小，把它拷贝至HDFS，并验证文件拷贝成功。

这一系列操作本身并不值得特别关注——它无非是第2章曾用过的hadoop fs命令的另一种用法。真正值得关注的是采用这种模式的原因。

尽管我们可以通过HTTP协议访问网络服务上的数据，但可直接使用的Hadoop工具都是基于文件的，并不支持这种远程的信息源。这就是我们需要先把网络数据拷贝到文件中，然后上传到HDFS的原因。

当然，我们可以通过第3章讲到的编程接口直接把数据写入HDFS，这种方法也会成功。但是，这种方法需要用户为每个不同的网络数据源编写定制的客户端。

一展身手

通过编程获取数据并把它写入HDFS是一种很强的能力，值得我们研究。**Apache HTTPClient**是一种非常流行的HTTP Java库，它是**HTTP Components**项目的一部分，其网址为http://hc.apache.org/httpcomponents-client-ga/index.html。

像刚才那样，使用HTTPClient和HDFS的Java接口获取网页，并把它写入HDFS。

10.3.1 把文件导入Hadoop

上例介绍了最简单的把文件数据导入Hadoop的方法，以及标准命令行工具或编程API的使用方法。本节没什么要特别讨论的内容，因为本书从始至终都在处理这种问题。

10.3.2 潜在的问题

尽管上述方法并无不妥，能够正常工作，但它们也存在一些问题，导致它们不适用于产品。

1. 把网络数据保留在网络上

先把网络数据拷贝到本地文件，然后把本地文件放到HDFS上的方法会影响系统的性能。由于要访问两次硬盘，而硬盘恰恰是系统中效率最低的部件，这就带来了额外的时延。如果一次调用可以获取大量数据，延时可能不是个大问题，但此时硬盘空间就成为需要考虑的问题。但对于多次少量数据的调用，延时就是个不得不考虑的问题。

2. 对Hadoop的依赖

对基于文件的方案而言，上述模型的一个隐含条件是，在我们访问文件的时候必须首先获知集群位置并获得Hadoop的访问权限。这就潜在地增加了系统依赖性——我们不得不把Hadoop集群的位置告诉负责下载网页文件的主机，它本来是不需要知道任何关于Hadoop集群信息的。有一个办法可以部分解决这个问题，那就是使用SFTP这样的工具把获取到的文件保存在Hadoop-aware主机上，然后再把它复制到HDFS。

3. 可靠性

读者可能会注意到，上述方法完全没有提到错误处理机制。我们使用的这些工具都没有实现内在的出错重试机制，这就意味着，我们需要在每次获取数据时都设计一种错误检查和重试机制。

4. 多此一举

这些特殊方法的最大问题在于，目前存在大量的实现类似任务的命令行工具和脚本。重复实现这些工具，以及复杂的错误跟踪所付出的的代价会随着时间日益凸显。

5. 一种通用的框架

做过企业计算化的人都知道，最好用通用集成框架来解决这个问题。这种思路是非常正确的，并且现在确实存在一款业界熟知的产品，它就是**EAI**（Enterprise Application Integration，企业应用集成）。

我们需要的就是一种有Hadoop知识并易于与Hadoop及相关项目整合的框架，无需用户投入大量精力编写定制的转换程序。用户可以自行实现这种框架，但接下来，我们将介绍**Apache Flume**，它提供了我们想要的大部分功能。

10.4 Apache Flume 简介

Flume是另一款与Hadoop紧密集成的Apache项目，其网址为http://flume.apache.org。本章剩余部分将主要学习该项目。

在介绍Flume功能之前，我们先要清楚它无法实现哪些功能。Flume被描述为分布式的日志

收集系统，也就是说，它处理的是基于文本行的文本数据。它并不是一个通用的分布式数据平台，特别是，别指望用它获取或传输二进制数据。

但是，由于Hadoop处理的绝大部分数据都与上述描述相符，很可能Flume可以满足读者的大部分数据获取需求。



Flume也不是一个通用的数据序列化框架，例如第5章曾用到的**Avro**、**Thrift**和 **Protocol Buffers**。我们将会看到，Flume设定了几种数据格式，除这些格式之外，它无法实现其他的数据序列化任务。

Flume可以从多个数据源获取数据，把这些数据传给远程主机（可能是一对多或流水线模型中的多个目标），再把它们传给多个目的端。尽管Flume提供了开发自定义数据源和数据目的端的编程API，但它原本就支持许多常见的场景。下面，我们通过安装使用Flume来学习其功能。

关于版本的说明

近期，Flume项目发生了几个主要的变动。原来的Flume（现在更名为**Flume OG**，意为原始版本）被**Flume NG**（Next Generation）所取代。尽管两个版本的基本原则和功能非常相似，但在实现上却存在较大差异。

因为Flume NG的使用会越来越广泛，本书将重点介绍它的相关知识。在一段时间内，Flume NG在某些方面不如Flume OG成熟，因此，如果Flume NG无法满足读者的某些需求，可以试试Flume OG。

10.5　实践环节：安装并配置 Flume

本节介绍Flume的下载、安装和配置。

(1) 从http://flume.apache.org/下载最新版的Flume NG二进制文件，把它保存到本地文件系统。

(2) 把文件复制到目标位置并解压。

```
$ mv apache-flume-1.2.0-bin.tar.gz /opt
$ tar -xzf /opt/apache-flume-1.2.0-bin.tar.gz
```

(3) 创建符号链接，指向Flume安装路径。

```
$ ln -s /opt/apache-flume-1.2.0 /opt/flume
```

(4) 定义环境变量FLUME_HOME。

```
Export FLUME_HOME=/opt/flume
```

(5) 把Flume安装路径下的bin子目录添加到用户路径中。

```
Export PATH=${FLUME_HOME}/bin:${PATH}
```

(6) 验证已设置了JAVA_HOME变量。

```
Echo ${JAVA_HOME}
```

(7) 验证已把Hadoop函数库路径加入到CLASSPATH变量中。

```
$ echo ${CLASSPATH}
```

(8) 在Hadoop目录下创建Flume的conf路径。

```
$ mkdir /home/hadoop/flume/conf
```

(9) 把所需的文件拷贝到刚创建的conf目录中。

```
$ cp /opt/flume/conf/log4j.properties /home/hadoop/flume/conf
$ cp /opt/flume/conf/flume-env.sh.sample /home/hadoop/flume/conf/
flume-env.sh
```

(10) 编辑flume-env.sh并设置JAVA_HOME。

原理分析

Flume的安装过程较为简单，与我们以前安装过的其他几个工具类似。

首先，获取Flume NG的最新版本（1.2.x及之后的版本都行）并保存到本地文件系统。然后把它复制到目标位置，解压缩并为了方便创建一个符号链接。

我们需要定义FLUME_HOME环境变量，并把安装目录下的bin文件夹添加到环境变量。和以前一样，这些设置都可以直接在命令行或脚本文件中实现。

Flume要求本机已设置JAVA_HOME环境变量，本例中我们通过echo命令来确认。Flume还要用到Hadoop函数库，因此需要检查CLASSPATH环境变量的值，确认其中已包含了Hadoop函数库的目录。

对于演示来讲，最后几个步骤并非必需的，但在产品集群中却会用到它们。Flume会搜索配置路径，其中包含定义了默认日志属性和环境变量的文件。我们发现，在正确设置配置路径的情况下，Flume会运行得更好，所以我们先创建了该目录，以后就无需改动了。

我们假设/home/hadoop/flume就是Flume的工作路径，Flume配置文件和其他文件将存储在该路径下。读者可根据自己系统的实际情况改变该路径。

使用Flume采集网络数据

上一节我们已安装了Flume，本节将用它采集一些网络数据。

10.6 实践环节：把网络流量存入日志文件

在第一个例子中，我们将使用简单的Flume配置，把采集到的网络数据存入主要的Flume日志文件。

(1) 把下列内容存入agent1.conf文件，并保存到Flume的工作目录下。

```
agent1.sources = netsource
agent1.sinks = logsink
agent1.channels = memorychannel

agent1.sources.netsource.type = netcat
agent1.sources.netsource.bind = localhost
agent1.sources.netsource.port = 3000


agent1.sinks.logsink.type = logger

agent1.channels.memorychannel.type = memory
agent1.channels.memorychannel.capacity = 1000
agent1.channels.memorychannel.transactionCapacity = 100

agent1.sources.netsource.channels = memorychannel
agent1.sinks.logsink.channel = memorychannel
```

(2) 启动一个Flume代理。

```
$ flume-ng agent --conf conf --conf-file 10a.conf    --name agent1
```

该命令的执行结果如下图所示。

```
File  Edit  View  Terminal  Help
hadoop@vm16:~/flume$ flume-ng agent --conf conf --conf-file 10a.conf --name agent1
Info: Sourcing environment configuration script /home/hadoop/flume/conf/flume-env.sh
Info: Including Hadoop libraries found via (/opt/hadoop/bin/hadoop) for HDFS access
Info: Excluding /opt/hadoop-1.0.3/libexec/../lib/slf4j-api-1.4.3.jar from classpath
Info: Excluding /opt/hadoop-1.0.3/libexec/../lib/slf4j-log4j12-1.4.3.jar from classpath
+ exec /opt/jdk/bin/java -Xmx20m -cp '/home/hadoop/flume/conf:/opt/flume/lib/*:/opt/hadoop-1.0.3/libexec/../conf:/opt/jdk/lib/tools.jar:/opt/hadoop-1.0.3/lib
exec/..:/opt/hadoop-1.0.3/libexec/../hadoop-core-1.0.3.jar:/opt/hadoop-1.0.3/libexec/../lib/asm-3.2.jar:/opt/hadoop-1.0.3/libexec/../lib/aspectjrt-1.6.5.jar:
/opt/hadoop-1.0.3/libexec/../lib/aspectjtools-1.6.5.jar:/opt/hadoop-1.0.3/libexec/../lib/avro-1.7.2.jar:/opt/hadoop-1.0.3/libexec/../lib/avro-mapred-1.7.2.ja
r:/opt/hadoop-1.0.3/libexec/../lib/commons-beanutils-1.7.0.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-beanutils-core-1.8.0.jar:/opt/hadoop-1.0.3/libexec/..
/lib/commons-cli-1.2.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-codec-1.4.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-collections-3.2.1.jar:/opt/hadoop-1.
0.3/libexec/../lib/commons-configuration-1.6.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-daemon-1.0.1.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-digester-
1.8.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-el-1.0.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-httpclient-3.0.1.jar:/opt/hadoop-1.0.3/libexec/../lib/co
mmons-io-2.1.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-lang-2.4.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-logging-1.1.1.jar:/opt/hadoop-1.0.3/libexec/.
./lib/commons-logging-api-1.0.4.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-math-2.1.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-net-1.4.1.jar:/opt/hadoop-
1.0.3/libexec/../lib/core-3.1.1.jar:/opt/hadoop-1.0.3/libexec/../lib/hadoop-capacity-scheduler-1.0.3.jar:/opt/hadoop-1.0.3/libexec/../lib/hadoop-fairschedule
r-1.0.3.jar:/opt/hadoop-1.0.3/libexec/../lib/hadoop-thriftfs-1.0.3.jar:/opt/hadoop-1.0.3/libexec/../lib/hsqldb-1.8.0.10.jar:/opt/hadoop-1.0.3/libexec/../lib/
jackson-core-asl-1.8.8.jar:/opt/hadoop-1.0.3/libexec/../lib/jackson-mapper-asl-1.8.8.jar:/opt/hadoop-1.0.3/libexec/../lib/jasper-compiler-5.5.12.jar:/opt/had
oop-1.0.3/libexec/../lib/jasper-runtime-5.5.12.jar:/opt/hadoop-1.0.3/libexec/../lib/jdeb-0.8.jar:/opt/hadoop-1.0.3/libexec/../lib/jersey-core-1.8.jar:/opt/ha
doop-1.0.3/libexec/../lib/jersey-json-1.8.jar:/opt/hadoop-1.0.3/libexec/../lib/jersey-server-1.8.jar:/opt/hadoop-1.0.3/libexec/../lib/jets3t-0.6.1.jar:/opt/h
adoop-1.0.3/libexec/../lib/jetty-6.1.26.jar:/opt/hadoop-1.0.3/libexec/../lib/jetty-util-6.1.26.jar:/opt/hadoop-1.0.3/libexec/../lib/jsch-0.1.42.jar:/opt/hado
op-1.0.3/libexec/../lib/junit-4.5.jar:/opt/hadoop-1.0.3/libexec/../lib/kfs-0.2.2.jar:/opt/hadoop-1.0.3/libexec/../lib/log4j-1.2.15.jar:/opt/hadoop-1.0.3/libe
xec/../lib/mockito-all-1.8.5.jar:/opt/hadoop-1.0.3/libexec/../lib/oro-2.0.8.jar:/opt/hadoop-1.0.3/libexec/../lib/paranamer-2.5.jar:/opt/hadoop-1.0.3/libexec/
../lib/servlet-api-2.5-20081211.jar:/opt/hadoop-1.0.3/libexec/../lib/xmlenc-0.52.jar:/opt/hadoop-1.0.3/libexec/../lib/jsp-2.1/jsp-2.1.jar:/opt/hadoop-1.0.3/l
ibexec/../lib/jsp-2.1/jsp-api-2.1.jar' -Djava.library.path=:/opt/hadoop-1.0.3/libexec/../lib/native/Linux-i386-32 org.apache.flume.node.Application --conf-fi
le 10a.conf --name agent1
```

(3) 在另一个窗口中，远程连接至本地主机的3000端口，然后输入一些文本。

```
$ curl telnet://localhost:3000

Hello
OK
Flume!
OK
```

(4) 使用Ctrl + C关闭curl连接。

(5) 查看Flume日志文件。

```
$ tail flume.log
```

该命令的执行结果如下所示。

```
2012-08-19 00:37:32,702 INFO sink.LoggerSink: Event: { headers:{}
body: 68 65 6C 6C 6F                                   Hello }
2012-08-19 00:37:32,702 INFO sink.LoggerSink: Event: { headers:{}
body: 6D 65                                              Flume }
```

原理分析

首先，我们在Flume工作目录下创建了一个配置文件。稍后会详细解释Flume的配置文件，但现在，假设Flume从一个称为信源（source）的部件接收数据，并把它写入称为信宿（sink）的目的地。

本例中，我们创建了一个**Netcat**信源，它监听某个端口上的网络连接。本例中，假设其监听的是本地主机的3000端口。

我们设置的信宿为`logger`类型，它会把输出写入一个日志文件。配置文件的剩余部分定义了一个名为`agent1`的代理，它会用到上述信源和信宿。

接着，我们使用`flume-ng`可执行文件启动一个Flume代理。我们将使用该工具启动所有Flume进程。请注意，该命令有下列几个选项：

- `agent`参数指定Flume启动一个代理，它泛指正在运行的涉及数据传输的Flume进程；
- `conf`路径，之前已提到过其含义；
- 针对待启动进程的特殊配置文件；
- 配置文件里设置的代理名称。

代理启动之后不久就会在屏幕上输出其运行结果。（很明显，我们可以根据产品集群的需要进行配置，并在后台运行该程序）。

在另一个窗口中，我们使用curl软件远程连接至本地主机的3000端口。打开这种远程连接会话的传统方式是使用`telnet`程序，但许多Linux系统中都默认安装了`curl`，基本上没人会再用`telnet`程序。

我们每行输入一个词，并按下回车键，然后使用“Ctrl + C”命令关闭远程会话。最后，我们查看位于Flume工作目录下的`flume.log`文件内容，从中发现了我们输入的所有单词。

10.7 实践环节：把日志输出到控制台

在某些情况下，查看日志文件的内容并不是很方便，尤其是在已经打开代理窗口的时候。接下来，我们修改设置，将日志同时输出到代理窗口。

(1) 启动Flume代理的时候新加一个参数。

```
$ flume-ng agent --conf conf --conf-file 10a.conf --name agent1
-Dflume.root.logger=INFO,console
```

该命令的执行结果如下所示。

```
Info: Sourcing environment configuration script /home/hadoop/
flume/conf/flume-env.sh
...
org.apache.flume.node.Application --conf-file 10a.conf --name
agent1

2012-08-19 00:41:45,462 (main) [INFO - org.apache.flume.lifecycle.
LifecycleSupervisor.start(LifecycleSupervisor.java:67)] Starting
lifecycle supervisor 1
```

(2) 在另一个窗口中，通过curl程序连接服务器。

```
$ curl telnet://localhost:3000
```

(3) 分两行输入Hello和Flume，按下“Ctrl + C”组合键，然后查看代理窗口。

```
File  Edit  View  Terminal  Help
7)] Node manager starting
2013-01-06 13:15:25,848 (lifecycleSupervisor-1-0) [INFO - org.apache.flume.lifecycle.LifecycleSupervisor.start(LifecycleSupervisor.java:67)] Starting lifecycle supervisor 8
2013-01-06 13:15:25,846 (lifecycleSupervisor-1-1) [INFO - org.apache.flume.conf.file.AbstractFileConfigurationProvider.start(AbstractFileConfigurationProvider.java:67)] Configuration provider starting
2013-01-06 13:15:25,850 (conf-file-poller-0) [INFO - org.apache.flume.conf.file.AbstractFileConfigurationProvider$FileWatcherRunnable.run(AbstractFileConfigurationProvider.java:195)] Reloading configuration file:10a.conf
2013-01-06 13:15:25,865 (conf-file-poller-0) [INFO - org.apache.flume.conf.FlumeConfiguration$AgentConfiguration.addProperty(FlumeConfiguration.java:988)] Processing:logsink
2013-01-06 13:15:25,867 (conf-file-poller-0) [INFO - org.apache.flume.conf.FlumeConfiguration$AgentConfiguration.addProperty(FlumeConfiguration.java:988)] Processing:logsink
2013-01-06 13:15:25,867 (conf-file-poller-0) [INFO - org.apache.flume.conf.FlumeConfiguration$AgentConfiguration.addProperty(FlumeConfiguration.java:902)] Added sinks: logsink Agent: agent1
2013-01-06 13:15:25,925 (conf-file-poller-0) [INFO - org.apache.flume.conf.FlumeConfiguration.validateConfiguration(FlumeConfiguration.java:122)] Post-validation flume configuration contains configuration  for agents: [agent1]
2013-01-06 13:15:25,927 (conf-file-poller-0) [INFO - org.apache.flume.conf.properties.PropertiesFileConfigurationProvider.loadChannels(PropertiesFileConfigurationProvider.java:249)] Creating channels
2013-01-06 13:15:26,018 (conf-file-poller-0) [INFO - org.apache.flume.instrumentation.MonitoredCounterGroup.<init>(MonitoredCounterGroup.java:68)] Monitoried counter group for type: CHANNEL, name: memorychannel, registered successfully.
2013-01-06 13:15:26,018 (conf-file-poller-0) [INFO - org.apache.flume.conf.properties.PropertiesFileConfigurationProvider.loadChannels(PropertiesFileConfigurationProvider.java:273)] created channel memorychannel
2013-01-06 13:15:26,039 (conf-file-poller-0) [INFO - org.apache.flume.sink.DefaultSinkFactory.create(DefaultSinkFactory.java:70)] Creating instance of sink: logsink, type: logger
2013-01-06 13:15:26,054 (conf-file-poller-0) [INFO - org.apache.flume.node.nodemanager.DefaultLogicalNodeManager.startAllComponents(DefaultLogicalNodeManager.java:92)] Starting new configuration:{ sourceRunners:{netsource=EventDrivenSourceRunner: { source:org.apache.flume.source.NetcatSource@1d902d3 }} sinkRunners:{logsink=SinkRunner: { policy:org.apache.flume.sink.DefaultSinkProcessor@4876e0 counterGroup:{ name:null counters:{} } }} channels:{memorychannel=org.apache.flume.channel.MemoryChannel@17480c9} }
2013-01-06 13:15:26,055 (conf-file-poller-0) [INFO - org.apache.flume.node.nodemanager.DefaultLogicalNodeManager.startAllComponents(DefaultLogicalNodeManager.java:99)] Starting Channel memorychannel
2013-01-06 13:15:26,059 (lifecycleSupervisor-1-0) [INFO - org.apache.flume.instrumentation.MonitoredCounterGroup.start(MonitoredCounterGroup.java:82)] Component type: CHANNEL, name: memorychannel started
2013-01-06 13:15:26,070 (conf-file-poller-0) [INFO - org.apache.flume.node.nodemanager.DefaultLogicalNodeManager.startAllComponents(DefaultLogicalNodeManager.java:127)] Starting Sink logsink
2013-01-06 13:15:26,073 (conf-file-poller-0) [INFO - org.apache.flume.node.nodemanager.DefaultLogicalNodeManager.startAllComponents(DefaultLogicalNodeManager.java:138)] Starting Source netsource
2013-01-06 13:15:26,073 (lifecycleSupervisor-1-3) [INFO - org.apache.flume.source.NetcatSource.start(NetcatSource.java:146)] Source starting
2013-01-06 13:15:26,093 (lifecycleSupervisor-1-3) [INFO - org.apache.flume.source.NetcatSource.start(NetcatSource.java:162)] Created serverSocket:sun.nio.ch.ServerSocketChannelImpl[/127.0.0.1:3000]
2013-01-06 13:16:12,135 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:70)] Event: { headers:{} body: 68 65 6C 6C 6F 0D                               hello. }
2013-01-06 13:16:12,136 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:70)] Event: { headers:{} body: 66 6C 75 6D 65 0D                               flume. }
```

原理分析

本例讲到的用法在调试或创建新数据流时非常有用。

从上例可以看出，默认情况下，Flume会把其日志写入保存在文件系统的文件内。更确切地

说，该动作是由conf路径下的log4j属性文件指定的。在某些情况下，我们想立即获得反馈，而无需经常查看日志文件或改动属性文件。

在命令行中明确设置flume.root.logger变量，就可以重写日志模块的默认配置，使Flume把输出直接发送给代理窗口。日志模块是标准的log4j，因此它支持常用的日志等级，例如DEBUG和INFO。

把网络数据写入日志文件

在默认情况下，Flume信宿会把收到的数据写入日志文件，这种方式有一些缺点，尤其是当我们想在其他程序中使用这些数据的时候。通过设置另一种信宿，我们可以把数据写入易用性更强的数据文件。

10.8 实践环节：把命令的执行结果写入平面文件

本节将介绍一种新的信源类型并在实践中使用它。

(1) 在Flume的工作目录下新建agent2.conf文件，并把下列内容保存到该文件中。

```
agent2.sources = execsource
agent2.sinks = filesink
agent2.channels = filechannel

agent2.sources.execsource.type = exec
agent2.sources.execsource.command = cat /home/hadoop/message

agent2.sinks.filesink.type = FILE_ROLL
agent2.sinks.filesink.sink.directory = /home/hadoop/flume/files
agent2.sinks.filesink.sink.rollInterval = 0

agent2.channels.filechannel.type = file
agent2.channels.filechannel.checkpointDir = /home/hadoop/flume/fc/
checkpoint
agent2.channels.filechannel.dataDirs = /home/hadoop/flume/fc/data

agent2.sources.execsource.channels = filechannel
agent2.sinks.filesink.channel = filechannel
```

(2) 在根目录下创建一个简单的测试文件。

```
$ echo "Hello again Flume!" > /home/hadoop/message
```

(3) 启动代理。

```
$ flume-ng agent --conf conf --conf-file agent2.conf --name agent2
```

(4) 在另一个窗口中，检查信宿文件的输出路径。

```
$ ls files
```

```
$ cat files/*
```

上述命令的结果如下所示。

```
hadoop@vm16:~/flume$ more /home/hadoop/message
Hello again Flume!
hadoop@vm16:~/flume$ ls files
1357479289746-1
hadoop@vm16:~/flume$ more files/1357479289746-1
Hello again Flume!
hadoop@vm16:~/flume$
```

原理分析

本例与前几个例子的工作流程类似。我们先为Flume代理创建一个配置文件，接着运行该代理，之后确认它抓到了我们想要的数据。

这次我们用的是exec信源和`file_roll`信宿。顾名思义，exec信源在主机上执行一个命令并将捕获的输出作为Flume代理的输入。虽然在上例中，只执行了一次命令，但这只是为了演示其原理。更为常见的情况是，代码中用到多条命令以生成持续的数据流。需要注意的是，读者可以对exec信宿进行配置，使其在命令终止的时候重启命令。

Flume代理的输出被写入配置文件的输出文件中。默认情况下，Flume每隔30秒钟将输出滚动写入到一个新文件中。为了便于在单个文件中跟踪Flume的输出，我们关闭了上述功能。

我们发现，输出文件中确实包含指定的`exec`命令的输出内容。

日志信宿与文件信宿

读者可能会对Flume中同时存在日志信宿和文件信宿感到困惑。从概念来讲，它们完成的任

务完全相同，那么区别在什么地方呢？

实际上，logger信宿比其他信宿更适合用作调试工具。它不仅仅记录Flume捕获的信源信息，同时加入了许多元数据和事件。然而，文件信宿准确地记录输入数据，因为这些数据在传给Flume时没有发生任何变化（如果需要修改数据，这也是可以实现的，稍后会看到这种情况）。

在大多数情况下，读者想用文件信宿捕获输入数据，但在开发过程中也可能会根据需要用到日志信宿。

10.9 实践环节：把远程文件数据写入本地平面文件

本节将展示另一个把数据写入文件信宿的例子。这次，我们会用到Flume的另一个功能——从远程客户端接收数据。

(1) 把下列内容保存到Flume工作目录下的`agent3.conf`文件中。

```
agent3.sources = avrosource
agent3.sinks = filesink
agent3.channels = jdbcchannel

agent3.sources.avrosource.type = avro
agent3.sources.avrosource.bind = localhost
agent3.sources.avrosource.port = 4000
agent3.sources.avrosource.threads = 5

agent3.sinks.filesink.type = FILE_ROLL
agent3.sinks.filesink.sink.directory = /home/hadoop/flume/files
agent3.sinks.filesink.sink.rollInterval = 0

agent3.channels.jdbcchannel.type = jdbc

agent3.sources.avrosource.channels = jdbcchannel
agent3.sinks.filesink.channel = jdbcchannel
```

(2) 创建新测试文件`/home/hadoop/message2`。

```
Hello from Avro!
```

(3) 启动Flume代理。

```
$ flume-ng agent -conf conf -conf-file agent3.conf -name agent3
```

(4) 在另一个窗口中，使用Flume Avro客户端向agent3发送文件。

```
$ flume-ng avro-client -H localhost -p 4000 -F /home/hadoop/
message
```

(5) 和以前一样，检查输出路径中的文件内容。

```
$ cat files/*
```

```
File  Edit  View  Terminal  Help
hadoop@vm16:~/flume$ more /home/hadoop/message2
Hello from Avro!
hadoop@vm16:~/flume$ more files/1357480848402-1
hadoop@vm16:~/flume$ flume-ng avro-client -H localhost -p 4000 -F /home/hadoop/message2
Warning: No configuration directory set! Use --conf <dir> to override.
Info: Including Hadoop libraries found via (/opt/hadoop/bin/hadoop) for HDFS access
Info: Excluding /opt/hadoop-1.0.3/libexec/../lib/slf4j-api-1.4.3.jar from classpath
Info: Excluding /opt/hadoop-1.0.3/libexec/../lib/slf4j-log4j12-1.4.3.jar from classpath
+ exec /opt/jdk/bin/java -Xmx20m -cp '/opt/flume/lib/*:/opt/hadoop-1.0.3/libexec/../conf:/opt/jdk/lib/tools.jar:/opt/hadoop-1.0.3/libexec/..:/opt/hadoop-1.0.
3/libexec/../hadoop-core-1.0.3.jar:/opt/hadoop-1.0.3/libexec/../lib/asm-3.2.jar:/opt/hadoop-1.0.3/libexec/../lib/aspectjrt-1.6.5.jar:/opt/hadoop-1.0.3/libexe
c/../lib/aspectjtools-1.6.5.jar:/opt/hadoop-1.0.3/libexec/../lib/avro-1.7.2.jar:/opt/hadoop-1.0.3/libexec/../lib/avro-mapred-1.7.2.jar:/opt/hadoop-1.0.3/libe
xec/../lib/commons-beanutils-1.7.0.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-beanutils-core-1.8.0.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-cli-1.2.jar
:/opt/hadoop-1.0.3/libexec/../lib/commons-codec-1.4.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-collections-3.2.1.jar:/opt/hadoop-1.0.3/libexec/../lib/commo
ns-configuration-1.6.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-daemon-1.0.1.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-digester-1.8.jar:/opt/hadoop-1.0.
3/libexec/../lib/commons-el-1.0.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-httpclient-3.0.1.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-io-2.1.jar:/opt/ha
doop-1.0.3/libexec/../lib/commons-lang-2.4.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-logging-1.1.1.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-logging-ap
i-1.0.4.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-math-2.1.jar:/opt/hadoop-1.0.3/libexec/../lib/commons-net-1.4.1.jar:/opt/hadoop-1.0.3/libexec/../lib/cor
e-3.1.1.jar:/opt/hadoop-1.0.3/libexec/../lib/hadoop-capacity-scheduler-1.0.3.jar:/opt/hadoop-1.0.3/libexec/../lib/hadoop-fairscheduler-1.0.3.jar:/opt/hadoop-
1.0.3/libexec/../lib/hadoop-thriftfs-1.0.3.jar:/opt/hadoop-1.0.3/libexec/../lib/hsqldb-1.8.0.10.jar:/opt/hadoop-1.0.3/libexec/../lib/jackson-core-asl-1.8.8.j
ar:/opt/hadoop-1.0.3/libexec/../lib/jackson-mapper-asl-1.8.8.jar:/opt/hadoop-1.0.3/libexec/../lib/jasper-compiler-5.5.12.jar:/opt/hadoop-1.0.3/libexec/../lib
/jasper-runtime-5.5.12.jar:/opt/hadoop-1.0.3/libexec/../lib/jdeb-0.8.jar:/opt/hadoop-1.0.3/libexec/../lib/jersey-core-1.8.jar:/opt/hadoop-1.0.3/libexec/../li
b/jersey-json-1.8.jar:/opt/hadoop-1.0.3/libexec/../lib/jersey-server-1.8.jar:/opt/hadoop-1.0.3/libexec/../lib/jets3t-0.6.1.jar:/opt/hadoop-1.0.3/libexec/../l
ib/jetty-6.1.26.jar:/opt/hadoop-1.0.3/libexec/../lib/jetty-util-6.1.26.jar:/opt/hadoop-1.0.3/libexec/../lib/jsch-0.1.42.jar:/opt/hadoop-1.0.3/libexec/../lib/
junit-4.5.jar:/opt/hadoop-1.0.3/libexec/../lib/kfs-0.2.2.jar:/opt/hadoop-1.0.3/libexec/../lib/log4j-1.2.15.jar:/opt/hadoop-1.0.3/libexec/../lib/mockito-all-1
.8.5.jar:/opt/hadoop-1.0.3/libexec/../lib/oro-2.0.8.jar:/opt/hadoop-1.0.3/libexec/../lib/paranamer-2.5.jar:/opt/hadoop-1.0.3/libexec/../lib/servlet-api-2.5-2
0081211.jar:/opt/hadoop-1.0.3/libexec/../lib/xmlenc-0.52.jar:/opt/hadoop-1.0.3/libexec/../lib/jsp-2.1/jsp-2.1.jar:/opt/hadoop-1.0.3/libexec/../lib/jsp-2.1/js
p-api-2.1.jar' -Djava.library.path=:/opt/hadoop-1.0.3/libexec/../lib/native/Linux-i386-32 org.apache.flume.client.avro.AvroCLIClient -H localhost -p 4000 -F
/home/hadoop/message2
hadoop@vm16:~/flume$ more files/1357480848402-1
Hello from Avro!
hadoop@vm16:~/flume$
```

原理分析

和以前一样，我们新建一个配置文件。这次，我们使用的是Avro信源。回忆一下，第5章曾讲到，Avro是一个数据序列化框架，也就是说，它负责对数据进行封包，并把数据从网络中的一个地方传到另一个地方。与Netcat信源类似，Avro信源对网络参数进行配置。本例中，它会监听本地主机的4000端口。我们配置代理使用文件信宿，之后启动该代理。

Flume既会用到Avro信源，也会用到独立的Avro客户端。后者可用于读取文件内容并把它发给网络上任何位置的Avro信源。在本例中，我们使用本地主机作为Avro信源，但要注意的是，Avro客户端需要准确知道Avro信源的主机名和端口号。因此，这并不是一个限制因素，Avro客户端可以把文件发给网络上任何位置的处于监听状态的Flume Avro信源。

Avro客户端读取文件内容，把它发给Flume代理，而代理会把这些数据写入文件信宿。通过检查输出文件的内容，我们确认Flume运作正常。

10.9.1　信源、信宿和信道

我们故意在前几个例子中使用了多种信源、信宿和信道，目的仅是为了说明组合使用这些部件。但是，我们没有详细研究它们，尤其是信道。现在，我们将深入研究这些内容。

1. 信源

我们已经学习了三种信源：Netcat，exec（可执行文件）以及Avro。Flume NG还支持使用一种序列生成器作为信源（主要用于测试），以及读取**syslogd**数据的信源（既支持TCP也支持UDP）。我们在代理中配置信源，信源在接收到足够数据可以生成一个Flume事件时，它会把新创建的事件发给信道。尽管信源的逻辑可能与它读取数据、转化事件以及处理故障的方式有关，但它却对事件的存储方式一无所知。信源负责把事件传给用户设置的信道，但如何处理事件却是对信源不可见的。

2. 信宿

除之前用到的logger和file_roll信宿之外，Flume还支持HDFS、HBase（两种类型）、Avro（用于代理链）、null（用于测试）和IRC（用于互联网中继聊天服务）这几种信宿。从概念上来讲，信宿与信源的概念正好相反。

信宿等着从信道接收事件，但它对信道的内部工作原理一无所知。在接收到数据后，信宿会把事件分发给各自的目标主机，并负责处理超时、重试以及循环之类的问题。

3. 信道

那么，这些连接着信源和信宿的神奇信道究竟是什么呢？就像名字和前面介绍的配置条目显示的那样，它们是负责传输事件的通信机制和保留机制。

当我们定义信源和信道时，它们在如何读取和写入数据方面有较大差异。例如，exec信源接收数据的速率远快于file_roll信宿的写入速度，或者信源有时需要暂停数据写入（例如在切换到新文件的时候，或处理系统I/O阻塞的时候）。因此，信道需要在信源和信道之间缓存数据，这样才能使数据最有效地流过代理。这也是配置文件的信道配置部分包括容量之类的配置元素的原因。

memory信道是最容易理解的一种信道类型，因为代理从信源把事件读入内存，然后传给信宿。但如果代理进程在该过程中途由于软件或硬件故障而崩溃，那么此时memory信道中的所有数据都会永久丢失。

我们用过的**file**和**JDBC**信道会永久存储事件，以防这种意外的数据丢失。在从信源读取事件之后，file信道将事件内容写入文件系统上的文件，只有代理成功将事件传给信宿后，该文件才会被删除。类似地，JDBC信道使用一个内嵌的Derby数据库以可恢复的形式存储事件。

这是一种典型的性能和可靠性之间的权衡问题。memory信道效率最高，但有数据丢失的风险。file和JDBC信道的效率相对较低，但能保证把数据传给信宿。具体选用哪种信道取决于应用程序以及每个事件的价值。



别在这个问题上太费心思。在实际使用中，答案通常是很明显的。同时也要仔细考虑用到的信源和信宿的可靠性。如果它们都靠不住，会丢失一些事件，那还有必要使用永久信道吗？

4. 定制信源、信道和信宿

读者可能会认为现有的信源、信道和信宿太少了，千万别这么想。Flume提供了一个自定义信源、信道、信宿的接口。另外，Flume OG的一部分组件还没有迁移到Flume NG中，但它们迟早会出现在Flume NG中的。

10.9.2　Flume配置文件

上一节，我们已经讨论了信源、信宿和信道。接下来，我们将详细研究一个以前用过的配置文件。

```
agent1.sources = netsource
agent1.sinks = logsink
agent1.channels = memorychannel
```

上述这些行指定了代理的名称，并定义了与之相关的信源、信宿和信道。每一行都可以有多个值，这些值以空格为分隔符。

```
agent1.sources.netsource.type = netcat
agent1.sources.netsource.bind = localhost
agent1.sources.netsource.port = 3000
```

上述行设置了信源的各个属性。因为我们使用了Netcat信源，配置项的值指定了绑定网络的方式。不同类型的信源有其自身独有的配置变量。

```
agent1.sinks.logsink.type = logger
```

这行指定了我们要用的logger信宿，其具体设置将通过命令行或log4j属性文件实现。

```
agent1.channels.memorychannel.type = memory
agent1.channels.memorychannel.capacity = 1000
agent1.channels.memorychannel.transactionCapacity = 100
These lines specify the channel to be used and then add the type
specific configuration values.    In this case we are using the memory
channel and we specify its capacity but - since it is non-persistent -
no external storage mechanism.
agent1.sources.netsource.channels = memorychannel
agent1.sinks.logsink.channel = memorychannel
```

最后几行设置了信源和信宿要用到的信道。尽管不同的代理要使用不同的配置文件，但为了简便也可以在一个配置文件中完成多个代理的设置，因为各个代理之间会实现必要的分隔。但是，这会使配置文件显得非常繁琐，可能会吓坏Flume的初学者。某个代理也可以包含多个流，例如，

我们可以把前两个例子合并到同一个配置文件和代理。

一展身手

动手实现它吧！把agent1和agent2合并为一个代理，并创建一个配置文件完成下列设置。

- 使用Netcat信源和logger信宿。
- 使用exec信源和file信宿。
- 分别为上述信源信宿对实现一个memory信道。

为了帮助读者顺利起步，作者给出一些定义作为示例。

```
agentx.sources = netsource execsource
agentx.sinks = logsink filesink
agentx.channels = memorychannel1 memorychannel2
```

10.9.3 一切都以事件为核心

在学习新例子之前，我们再讨论一个概念。究竟什么是事件呢？

回忆一下，Flume显然以日志文件为基础，那么在大多数情况下，事件就相当于一行接一行的文本。我们曾在信源和信宿中看到过这样的数据。

但是，事件并非全部是文本数据。例如，UDP syslogd信源把接收到的每个数据包当做一个事件，并在系统中传输。在使用这种类型的信源和信宿时，对事件的定义保持不变，而当读取文件时，我们不得不使用基于文本行的事件概念。

10.10 实践环节：把网络数据写入 HDFS

在专门研究Hadoop的书中讨论Flume，但至今为止却尚未介绍如何在Hadoop中使用Flume，多少有点不太合适。接下来，我们将要介绍如何把数据写入HDFS。

(1) 在Flume工作目录下新建agent4.conf文件，并把以下内容保存到该文件中。

```
agent4.sources = netsource
agent4.sinks = hdfssink
agent4.channels = memorychannel

agent4.sources.netsource.type = netcat
agent4.sources.netsource.bind = localhost
agent4.sources.netsource.port = 3000

agent4.sinks.hdfssink.type = hdfs
agent4.sinks.hdfssink.hdfs.path = /flume
agent4.sinks.hdfssink.hdfs.filePrefix = log
agent4.sinks.hdfssink.hdfs.rollInterval = 0
```

```
agent4.sinks.hdfssink.hdfs.rollCount = 3
agent4.sinks.hdfssink.hdfs.fileType = DataStream

agent4.channels.memorychannel.type = memory
agent4.channels.memorychannel.capacity = 1000
agent4.channels.memorychannel.transactionCapacity = 100

agent4.sources.netsource.channels = memorychannel
agent4.sinks.hdfssink.channel = memorychannel
```

(2) 启动代理。

```
$ flume-ng agent -conf conf -conf-file agent4.conf -name agent4
```

(3) 在另一个窗口中，开启一个远程连接并向Flume发送7个事件。

```
$ curl telnet://localhost:3000
```

(4) 查看输出目录中的文件，并检查文件内容。

```
$ hadoop fs -ls /flume
$ hadoop fs -cat "/flume/*"
```

上述命令的执行结果如下图所示。

```
hadoop@vm16: ~
File Edit View Terminal Help
hadoop@vm16:~$ curl  telnet://localhost:3000
Hello
OK
Hadoop
OK
how
OK
are
OK
you
OK
today?
OK
Bye!
OK
^C
hadoop@vm16:~$ hadoop fs -ls /flume
Found 3 items
-rw-r--r--   1 hadoop supergroup         17 2013-01-06 14:37 /flume/log-.1357483063181
-rw-r--r--   1 hadoop supergroup         15 2013-01-06 14:37 /flume/log-.1357483063182
-rw-r--r--   1 hadoop supergroup          0 2013-01-06 14:37 /flume/log-.1357483063183.tmp
hadoop@vm16:~$ hadoop fs -cat "/flume/*"
Hello
Hadoop
how
are
you
today?
Bye!
hadoop@vm16:~$
```

原理分析

这次，我们使用的是Netcat信源和HDFS信道。从配置文件中可以看出，我们需要设置文件位置、文件前缀以及从某个文件切换到另一个文件的策略等内容。本例中，我们指明文件位于`/flume`目录下，每个文件都以`log-`为前缀，并且每个文件最多只能存储3条数据（很明显，这么少的数据只能用于测试）。

在启动代理后，我们再次使用curl向Flume发送7个事件，每个事件仅包含1个单词。接着，我们使用Hadoop命令行工具查看/flume目录并验证输入数据被写入HDFS。

请注意，第三个HDFS文件扩展名为`.tmp`。回忆一下，我们设置的是每个文件包含3条记录，但仅输入7个值。因此，有2个文件已达到其数据容量，而第3个文件则刚刚开始。Flume用`.tmp`后缀标记正在写入的文件，这就一眼能区分出完整文件和正在写入的文件，而MapReduce作业只处理完整文件。

10.11 实践环节：加入时间戳

前面我们曾提到Flume提供了一些机制，用于写入稍微复杂的文件数据。本节，我们将展示一些很普通的做法——把动态生成的时间戳和数据一同写入文件。

(1) 把下列配置内容保存为`agent5.conf`文件。

```
agent5.sources = netsource
agent5.sinks = hdfssink
agent5.channels = memorychannel

agent5.sources.netsource.type = netcat
agent5.sources.netsource.bind = localhost
agent5.sources.netsource.port = 3000
agent5.sources.netsource.interceptors = ts

agent5.sources.netsource.interceptors.ts.type = org.apache.flume.
interceptor.TimestampInterceptor$Builder

agent5.sinks.hdfssink.type = hdfs
agent5.sinks.hdfssink.hdfs.path = /flume-%Y-%m-%d
agent5.sinks.hdfssink.hdfs.filePrefix = log-
agent5.sinks.hdfssink.hdfs.rollInterval = 0
agent5.sinks.hdfssink.hdfs.rollCount = 3
agent5.sinks.hdfssink.hdfs.fileType = DataStream

agent5.channels.memorychannel.type = memory
agent5.channels.memorychannel.capacity = 1000
agent5.channels.memorychannel.transactionCapacity = 100

agent5.sources.netsource.channels = memorychannel
```

```
agent5.sinks.hdfssink.channel = memorychannel
```

(2) 启动代理。

```
$ flume-ng agent -conf conf -conf-file agent5.conf -name agent5
```

(3) 在另一个窗口中，开启一个远程连接会话，并向Flume发送7个事件。

```
$ curl telnet://localhost:3000
```

(4) 查看HDFS上的输出文件。

```
$ hadoop fs -ls /
```

代码输出如下所示。

```
File  Edit  View  Terminal  Help
hadoop@vm16:~$ curl  telnet://localhost:3000
Yet
OK
another
OK
example
OK
of
OK
data
OK
onto
OK
HDFS
OK
^C
hadoop@vm16:~$ hadoop fs -ls /
Found 4 items
drwxr-xr-x   - hadoop supergroup          0 2013-01-06 14:52 /flume-2013-01-06
drwxr-xr-x   - hadoop supergroup          0 2013-01-05 23:30 /tmp
drwxr-xr-x   - garry  supergroup          0 2013-01-05 23:29 /user
drwxr-xr-x   - hadoop supergroup          0 2013-01-05 21:39 /var
hadoop@vm16:~$ hadoop fs -ls /flume-2013-01-06
Found 3 items
-rw-r--r--   1 hadoop supergroup         20 2013-01-06 14:52 /flume-2013-01-06/log-.1357483958978
-rw-r--r--   1 hadoop supergroup         13 2013-01-06 14:52 /flume-2013-01-06/log-.1357483958979
-rw-r--r--   1 hadoop supergroup          0 2013-01-06 14:52 /flume-2013-01-06/log-.1357483958980.tmp
hadoop@vm16:~$
```

原理分析

我们对上一个配置文件进行了一些改动，为Netcat信源新加一个`interceptor`属性，并指定该属性值为`TimestampInterceptor`。

Flume拦截器其实就是一些插件，它们可以在从信源向信宿传输事件的过程中操作和修改事件。大多数拦截器要么在原事件的基础上加入一些元数据（本例即是这种情况），要么基于某些规则删去一些事件。除了几个内置的拦截器外，Flume还提供了用户自定义拦截器的机制。

本例中我们使用**timestamp**拦截器，它在事件元数据的基础上附加了读取事件时的Unix时间戳。通过这种方式，我们扩展了写入事件的HDFS路径的名称。

前几个例子中，我们只是把所有事件写入`/flume`路径，本例中，我们把事件的写入路径设置为`/flume-%Y-%m-%d`。在运行代理并向Flume发送一些数据之后，我们查看HDFS，发现事件输出路径带有“年/月/日”这样的词缀。

HDFS信宿支持许多其他变量，例如信源的主机名以及其他临时变量，使用这些临时变量可以以秒为单位对数据进行准确分块。

拦截器的功能非常明显。之前，代理将所有事件写入同一个信宿路径，这样随着时间推移，该目录变得越来越大。而使用拦截器之后，不仅可以实现数据的自动分块，简化了数据管理，而且便于MapReduce作业使用这些数据。例如，假如MapReduce作业每次处理的都是一小时内的数据，那么就可以在Flume中把输入事件按小时分块，写入不同路径，这样就会极大简化该过程。

准确的说，拦截器为Flume中传输的事件添加了完整的Unix时间戳，也就是说，该时间戳可以精确到秒。本例中，我们在路径名中只用到了和日期相关的部分，如果读者需要以小时或更细粒度对数据进行分块，那么就会用到与时间相关的变量。



上述内容假设处理事件时的时间戳足以满足读者需求。假如一批文件同时被处理并反馈到Flume系统，那么文件数据的时间戳应该是另一个时间，而不是它们被处理的时间。在此情况下，读者需要编写自定义拦截器基于文件内容设置时间戳。

使用Sqoop还是使用Flume

如果读者需要把关系数据库中的数据导出到HDFS上，那么Sqoop和Flume这两个工具哪个更合适？我们已经了解了Sqoop如何执行导出任务，使用Flume也可以完成类似任务：既可以编写自定义代码也可以在exec信源中调用`mysql`命令。

最好根据数据类型选择。要导出的数据是日志数据吗？还是更复杂的数据？

在很大程度上，Flume被设计用于处理日志数据，事实证明，它擅长处理这类数据。但在大多数情况下，Flume网络负责把事件从信源传到信宿，却不会真正传输日志数据本身。如果用户在多个关系数据库中存有日志数据，使用Flume应该是个不错的选择，尽管我会怀疑数据库是否有能力长期存储日志记录。

非日志数据可能需要执行一些只有Sqoop才能完成的数据操作。上一章我们使用Sqoop完成的许多数据转换，例如指定目标列的子集，无法使用Flume完成。还有，如果用户需要处理结构化数据的各个字段，单独使用Flume也无法完成这个任务。如果读者想要与Hive集成，那么此时Sqoop是唯一的选择。

当然，请记住，这些工具还可以协同处理更复杂的任务。我们可以使用Flume把事件汇聚到HDFS，使用MapReduce进行处理，然后通过Sqoop导出到一个关系数据库中。

10.12　实践环节：多层 Flume 网络

本节我们将实践如何使用一个Flume代理作为另一个代理的信宿。

(1) 把下列内容保存到agent6.conf文件。

```
agent6.sources = avrosource
agent6.sinks = avrosink
agent6.channels = memorychannel

agent6.sources.avrosource.type = avro
agent6.sources.avrosource.bind = localhost
agent6.sources.avrosource.port = 2000
agent6.sources.avrosource.threads = 5

agent6.sinks.avrosink.type = avro
agent6.sinks.avrosink.hostname = localhost
agent6.sinks.avrosink.port = 4000

agent6.channels.memorychannel.type = memory
agent6.channels.memorychannel.capacity = 1000
agent6.channels.memorychannel.transactionCapacity = 100

agent6.sources.avrosource.channels = memorychannel
agent6.sinks.avrosink.channel = memorychannel
```

(2) 按照agent3.conf的配置启动一个代理，也就是说，使用Avro信源和file信宿。

```
$ flume-ng client -conf conf -conf-file agent3.conf agent3
```

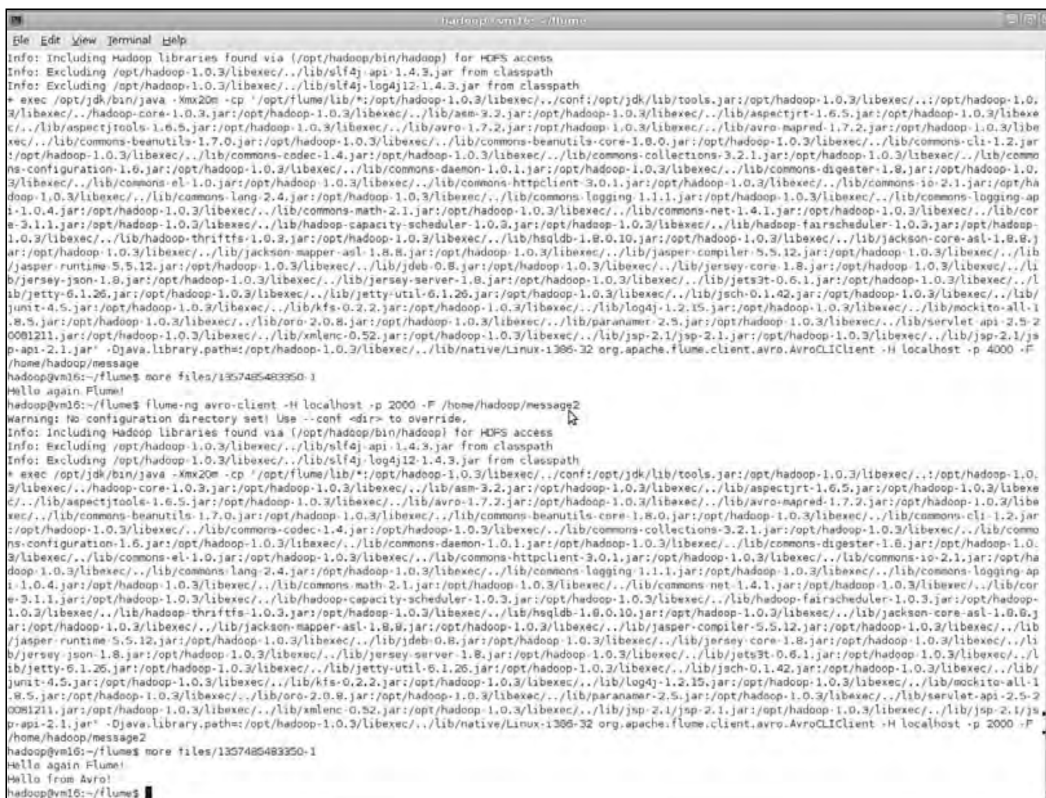
(3) 在第二个窗口中，按照agent6.conf的配置启动另一个代理。

```
$ flume-ng client -conf conf -conf-file agent6.conf agent6
```

(4) 再打开一个窗口，使用Avro客户端分别向上述两个代理发送一个文件。

```
$ flume-ng avro-client -H localhost -p 4000 -F /home/hadoop/
message
$ flume-ng avro-client -H localhost -p 2000 -F /home/hadoop/
message2
```

(5) 检查输出路径，确认输出文件存在于该目录下。



原理分析

首先，我们定义了一个新代理，其信源和信宿都是Avro类型。之前从未用过Avro信宿，它不会把事件写入本地文件或HDFS文件，而是把事件发给远程Avro信源。

我们启动agent6代理之后，又启动了一个前几节用过的代理agent3。回忆一下，agent3使用Avro信源和file_roll信宿。我们将第一个代理的Avro信宿指向第二个代理的Acro信源的主机地址和端口，通过这种方式构建了一个数据链路。

在agent6和agent3都运行起来后，我们使用Avro客户端向每个代理发送一个文件，并确认这两个文件都存在于agent3信宿指定的目录下。

能达到这个效果并非只是技术原因使然。更重要的原因在于，Flume支持用户构建非常复杂的分布式事件收集网络。我们可以认为不同类型的多个代理可以把事件传给链条中的下一个代理，它们就像是事件汇集点一样。

10.13 实践环节：把事件写入多个信宿

我们还希望构建这样的网络，一个代理可以向多个信宿写入数据。本节将实现它。

(1) 把下列内容保存为agent7.conf文件。

```
agent7.sources = netsource
agent7.sinks = hdfssink filesink
agent7.channels = memorychannel1 memorychannel2

agent7.sources.netsource.type = netcat
agent7.sources.netsource.bind = localhost
agent7.sources.netsource.port = 3000
agent7.sources.netsource.interceptors = ts

agent7.sources.netsource.interceptors.ts.type = org.apache.flume.
interceptor.TimestampInterceptor$Builder

agent7.sinks.hdfssink.type = hdfs
agent7.sinks.hdfssink.hdfs.path = /flume-%Y-%m-%d
agent7.sinks.hdfssink.hdfs.filePrefix = log
agent7.sinks.hdfssink.hdfs.rollInterval = 0
agent7.sinks.hdfssink.hdfs.rollCount = 3
agent7.sinks.hdfssink.hdfs.fileType = DataStream

agent7.sinks.filesink.type = FILE_ROLL
agent7.sinks.filesink.sink.directory = /home/hadoop/flume/files
agent7.sinks.filesink.sink.rollInterval = 0

agent7.channels.memorychannel1.type = memory
agent7.channels.memorychannel1.capacity = 1000
agent7.channels.memorychannel1.transactionCapacity = 100

agent7.channels.memorychannel2.type = memory
agent7.channels.memorychannel2.capacity = 1000
agent7.channels.memorychannel2.transactionCapacity = 100

agent7.sources.netsource.channels = memorychannel1 memorychannel2
agent7.sinks.hdfssink.channel = memorychannel1
agent7.sinks.filesink.channel = memorychannel2

agent7.sources.netsource.selector.type = replicating
```

(2) 启动代理agent7。

```
$ flume-ng agent -conf conf -conf-file agent7.conf -name agent7
```

(3) 开启一个远程会话连接并向Flume发送一个事件。

```
$ curl telnet://localhost:3000
```

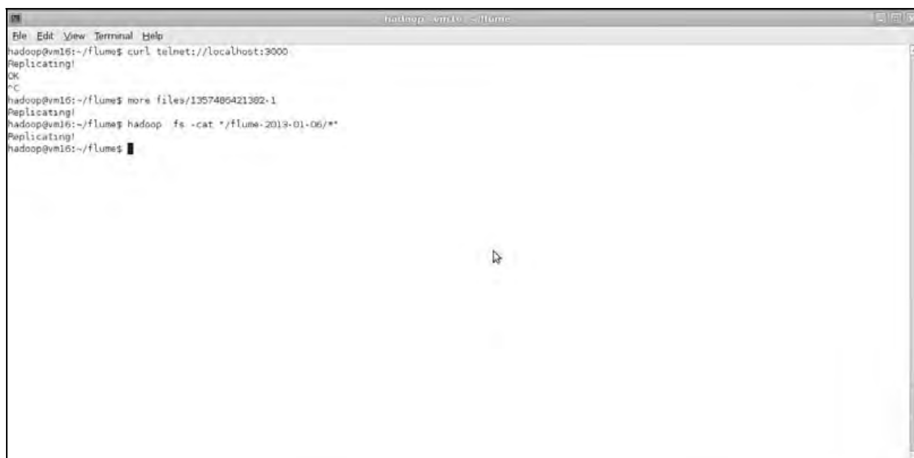
上述命令的执行结果如下所示。

```
Replicating!
```

检查HDFS上的输出目录及文件信宿的内容。

```
$ cat files/*
$ hdfs fs -cat "/flume-*/*"
```

上述命令的执行结果如下图所示。



原理分析

我们创建了一个配置文件，该文件设置了一个Netcat信源，以及两个信宿，分别是file信宿和HDFS信宿。同时，还设置了两个memory信道分别连接信源和两个信宿。

接着，我们设置信源选择器的类型为`replicating`，也就是说，所有事件都会同时发送给上述两个信道。 像往常一样启动代理并向信源发送一个事件之后，我们确认，该事件确实被同时写入文件系统和HDFS信宿。

10.13.1　选择器的类型

信源选择器有两种工作模式，一种是replicating模式，另一种是multiplexing模式。**multiplexing**模式的信源选择器会依据事件的特定头部字段值判断向哪个信道发送事件。

10.13.2　信宿故障处理

信宿作为数据输出端，本质上就决定了它可能随着时间推移会逐渐发生故障。就像其他输入输出设备一样，信宿也会遇到写入速度达到上限、存储空间已用尽或者脱机离线的问题。

刚才已看到，Flume允许信源使用选择器实现事件复制或复用。与此类似，Flume提出了信宿处理器的概念。

Flume定义了两种信宿处理器，它们分别是故障恢复（failover）信宿处理器和负载均衡（load balancing）信宿处理器。

信宿处理器把所有信宿看做一个信宿组，它会依据各个信宿的类型在事件到达时采取不同的措施。负载均衡信宿处理器每次向信宿发送一个事件，它采用轮询（round-robin）或随机算法选择下次使用的信宿。如果某个信宿出现故障，信宿处理器会向另一个信宿发送相同事件，但发生故障的信宿仍然保留在信宿池中。

与此不同，故障恢复信宿处理器把所有信宿视为一个优先表，只有高优先级信宿发生故障后，它才会使用低优先级信宿。故障恢复信宿处理器会从优先表中删除发生故障的信宿，在经过一段冷却期后重试该信宿故障是否修复，以避免后续的大量故障。

一展身手：信宿故障处理

创建一个包含3个HDFS信宿的Flume配置文件，每个信宿对应着HDFS上的不同位置。使用负载均衡信宿处理器，确认事件被平均写到每个信宿，然后使用故障恢复信宿处理器，展示各个信宿的优先级。

你能想办法让代理选用指定信宿代替优先级最高的信宿吗？

10.13.3　使用简单元件搭建复杂系统

目前，我们已经学习了Flume的大部分关键功能。以前曾提到，Flume是一个框架，我们应当认真考虑这个说法。Flume的部署方式非常灵活，我们曾介绍过的其他产品都无法与之相比。

Flume的灵活性来源于其中一小部分功能。通过信道连接信源和信宿，并且支持多个代理和多个信道，这就是Flume灵活性的一种具体表现。这些功能看上去没什么了不起，但这些模块却可以用来构建下述系统，该系统能把多个网页服务器群的日志汇聚到一个Hadoop集群。

- 服务器群中的每个节点都运行一个代理，负责依次获取所有的本地日志文件。
- 这些日志文件被发送给一个高可靠性的汇聚点。每个服务器群中都有一个汇聚点，负责执行一些处理任务，并在原事件基础上附加一些元数据，把这些记录分成3类。
- 第一级集合器把事件发给能够访问Hadoop集群的一个代理。集合器提供了多个访问点，1类事件和2类事件被发给第一级聚合器，3类事件被发给第二级聚合器。
- 最后一级聚合器把1类事件和2类事件写入HDFS的不同位置，同时2类事件也被写入到本地文件系统。3类事件直接写入HBase。

简单的元件就能组合搭建这么复杂的系统，简直太神奇了！

一展身手：使用简单元件搭建复杂系统

作为一个练习题，考虑一下如何实现上述系统，并确定流程中每一步要用到怎样的Flume配置。

10.14 更高的视角

读者需要意识到，用户不光要考虑“简单地”从某个节点向另一个节点传输数据。最近，由于某些原因，数据的生命周期管理（data lifecycle management）这一概念被广为使用。接下来，在系统出现数据泛滥的情况之前，我们将简要介绍一些需要用户考虑的问题。

10.14.1 数据的生命周期

关于数据生命周期的主要问题是，数据要存储多久其价值才能超过存储成本。永久存储数据看似极具吸引力，但随着时间推移，存储的数据量越来越大，存储成本也会急剧上升。这些成本不单单是金钱方面的成本。随着数据量的增长，许多系统的性能也会逐步降低。

关于数据存储多长时间最为合适，这一问题的答案很少由技术因素来决定。相反，数据价值以及业务成本才是决定性因素。有些时候，用户很快就会发现数据毫无价值。在另一些情况下，出于竞争或法律原因，业务无法删除这些数据。确定数据在业务流程中所处地位，然后采取相应措施。

当然，一定要记住，保留或删除数据并不是一个非此即彼的问题。用户还可以随着数据保留时间的增长，将其逐步迁移到成本较低、性能较差的多层存储系统中。

10.14.2 集结数据

另一方面，读者通常需要考虑数据是如何送入MapReduce这类数据处理平台的。由于使用了多个数据源，用户往往希望把所有数据都放在同一个大容量存储系统中。

正如我们之前所看到的，Flume可以用参数表示HDFS上的数据写入位置，有助于解决这个问题。但是，最好把这些数据写入位置视为临时集结区，Flume会先把数据写入这个位置再进行处理。在数据处理结束之后，这些数据会被移到长期目录结构。

10.14.3 调度

在很多情况下，我们曾提到过，Flume可能需要一个外部任务执行某些操作。如前所述，一旦文件写入HDFS，我们就想用Flume对其进行处理，但是该任务是如何调度执行的呢？或者

说，我们如何进行后期处理，如何对数据进行存档或删除老数据，甚至如何删除源主机上的日志文件？

Linux系统提供的**logrotate**工具可以调度上述部分任务，例如源主机上的日志删除工作，但读者需要自行创建工具实现完成其他任务的调度工作。类似**cron**这样的常用工具就能完成这些功能，但随着系统复杂性的提升，用户需要寻找其他更复杂的调度系统。下一章，我们将会简要介绍这种系统与Hadoop的集成。

10.15　小结

本章讨论了如何获取网络上的数据，并使用Hadoop来处理这些数据。我们发现，实际上这是一个很常见的问题。尽管我们可以使用Hadoop的专属工具，如Flume，但这并不是唯一的解决方案。我们特别对可能写入Hadoop的数据进行了分类，通常把它们分为网络数据和文件数据。我们介绍了一些使用命令行工具获取数据的方法。尽管这些方法确实有效，但它们太过简单，并不适合更复杂的情况。

我们把Flume视为一个可灵活使用的框架，它可以定义和管理数据（尤其是日志文件）路径，Flume系统认为数据首先达到信源，经过信道处理后，被写入信宿。

接着，我们学习了很多Flume功能，包括如何使用不同类别的信源、信道和信宿。从中可以学到，如何把简单的功能模块组合成非常复杂的系统，最后我们以对数据管理的更普遍性的思考结束本章内容。

全书主要内容到此结束。下一章我们将简要介绍读者可能感兴趣的大量其他项目，并强调了一些加入Hadoop开发者社区和获得帮助的方法。

第11章 展望未来

正如书名所述，本书目的在于向初学者深入介绍Hadoop相关知识及其应用。读者经常会发现，Hadoop生态系统的范围远远不止这些核心产品。本章我们将快速浏览一些有趣的应用。

本章包括以下内容：

- 总结本书涵盖的内容；
- 本书未涉及的内容；
- 即将发生的Hadoop变革；
- 其他版本的Hadoop软件安装包；
- 其他重要的Apache项目；
- 其他程序设计方法；
- 信息来源及帮助。

11.1 全书回顾

因为我们把读者群设定为Hadoop初学者，所以希望初学者从书中能学到Hadoop的核心概念和工具，为今后的学习和工作打下坚实基础。而且，我们设计了一些实践环节，有助于读者把Hadoop集成到原有系统架构中。

尽管起初Hadoop只是一个独立的产品，但毫不客气地说，近年来以Hadoop为中心的生态系统呈爆发趋势。读者还可以选用其他版本的软件包部署Hadoop，有些版本的软件包提供了定制的商用扩展功能。目前，以Hadoop为基础的相关项目和应用实在是太多了，它们要么实现了某个新功能，要么以其他方式实现了现有功能。赶快使用Hadoop吧，这真是一个激动人心的时刻，快来看看还有些什么好玩的东西。



当然，对Hadoop生态系统的概述由作者的兴趣和偏好决定，可能并不全面，而且涉及的相关技术和应用在写作时也已过时。换句话说，千万别认为书中讲到的东西现在全都能用，权当是开胃菜好了。

11.2 即将到来的 Hadoop 变革

在讨论其他版本的Hadoop软件安装包之前，我们先来看看Hadoop将要发生的一些变革。我们曾提到过Hadoop 2.0的变化，主要是使用新的BackupNameNode和CheckpointNameNode服务提高了NameNode的可用性。这是Hadoop的一个重大变革，因为它能增强HDFS的健壮性，极大地提高企业信用，并能实现集群操作的流水化。再怎么夸大NameNode HA对Hadoop的影响都不为过，几年之后，人们甚至会想，没发明NameNode HA技术之前，Hadoop是如何运作的。

发生这些变革的同时，MapReduce也不是一成不变的。实际上，我们介绍的这些Hadoop变革不可能带来太多的直接影响，却能带来根本上的改变。

最初，开发者把改进后的Hadoop称为**MapReduce 2.0**或**MRV2**。但是，现在我们认为**YARN**（Yet Another Resource Negotiator）这个名称更为贴切，因为主要变化发生在Hadoop平台而非MapReduce。YARN的目标是在Hadoop基础上构建一个集群资源分配平台，可以为某些应用分配集群资源，而MapReduce只是这些应用中的一种。

目前，JobTracker负责两项不同的任务：管理某一MapReduce作业的进度（也要确定任意时刻处于空闲状态的集群资源），以及在作业的不同阶段分配资源。YARN把这些任务分给了几个独立的组件。一个全局的ResourceManager负责集群资源管理，它通过每台主机上的NodeManager实现该功能；一个独立的**ApplicationManager**，它与ResourceManager通信以获取作业所需资源。

YARN中的MapReduce接口保持不变，所以从客户的角度来看，所有现有代码都能在新平台上运行。但是由于新开发了ApplicationManager，我们更倾向于把Hadoop视为支持多种处理模型的通用任务处理平台。早期已迁移到YARN的处理模型包括基于流的处理模型，以及**MPI**（Message Passing Interface，消息传递接口）的一个端口，它在科学计算中得到了广泛使用。

11.3 其他版本的 Hadoop 软件包

回顾第2章内容，我们从Hadoop主页下载其安装包。但是这并非获得Hadoop的唯一方式，读者可能对此感到奇怪。更让人奇怪的是，大多数生产环境中使用的Hadoop并不是Apache提供的版本。

为什么会有其他版本

Hadoop是一款开源软件。任何遵守Apache软件许可证的人都可以发布自己开发的Hadoop版本。人们主要出于两个考虑来创建其他Hadoop版本。

1. 打包

有些人开发这些安装包是为了捆绑其他软件，例如Hive、HBase、Pig，还有许多其他项目。大多数项目的安装方法差异很大（HBase是个例外，事实证明它的手动安装更为困难），一些细微的版本不兼容问题只有在系统处理特殊的任务时才会显现。把这些软件打包发布可以提供一组兼容的软件。

2. 免费的和商用的扩展

作为一个开源项目，Hadoop软件包的发布相对比较自由，开发人员可以自由使用私有扩展增强Hadoop，使其既可以成为免费开源产品也可以成为商业产品。

这是一个有争议的问题，一些开源支持者不希望把任何成功的开源产品进行商业化运作。在他们看来，商业公司只是在攫取开源社区的劳动果实，无须自主创建这些软件即可不劳而获。另一些人认为这对Apache许可很有好处，基础产品永远是免费的，个人用户和商业公司可以选择是否使用商用扩展。我们不对这些观点进行任何评论，但我们时刻面对着这个争议。

理解了不同版本软件包的由来之后，接下来我们将选取几个较受欢迎的软件版本进行介绍。

- Cloudera开发的Hadoop版本

Cloudera开发的Hadoop安装包是目前使用最广泛的Hadoop安装包，以后简写为**CDH**（Cloudera Distribution for Hadoop）。回忆一下，Sqoop即是由Cloudera公司开发的，然后将其捐献给了开源团体，Doug Cutting现在为这家公司效力。

读者可通过http://www.cloudera.com/hadoop了解 Cloudera版Hadoop，同时该页面还包含大量Apache产品，如Hive、Pig、HBase、 Sqoop和Flume, 还有其他一些不太出名的产品，如Mahout和Whir。我们稍后会介绍这些产品。

CDH提供了多种格式的安装包供用户下载，并以即装即用的方式部署软件。例如，基本的Hadoop产品被分成多个不同的安装包，分别对应着NameNode、TaskTracker等Hadoop组件，每个组件都集成了标准的Linux基础服务。

CDH是第一个被广泛应用的可选安装包，它集成了很多实用软件，用户不仅可免费使用，而且质量可靠，因此很多用户都选择了这个版本的Hadoop。

另外，Cloudera也提供了其他商业产品，如Hadoop管理工具、Hadoop培训、技术支持和咨询服务。详细内容参见公司主页的介绍。

- Hortonworks推出的数据平台

2011年，Yahoo公司内部负责大量Hadoop开发工作的部门被独立出来，成立了一个名为**Hortonworks**的新公司。他们也推出了自行研发的预集成的Hadoop安装包，取名为**Hortonworks Data Platform**（简称HDP），网址为 http://hortonworks.com/products/hortonworksdataplatform/。

HDP的概念与CDH相似，但这两个产品的侧重点有所不同。HDP，甚至包括其管理工具在内，基本上是一个完全开源的产品。HDP还提供对Talend Open Studio的支持，Hortonworks公司把它定位成一个关键的集成平台。Hortonworks公司不提供任何商用软件，它主要通过提供专业服务和技术支持实现盈利。

Cloudera和Hortonworks都是拥有重要工程技术的风险企业，很多对Hadoop有杰出贡献的开发人员都受雇于这两个公司。但是，它们的技术基础都是这些相同的Apache项目，区别在于软件集成的方式不同，使用的软件版本不同，以及各公司提供的增值服务不同。

- MapR

MapR Technologies公司提供了一种全新的安装包，通常人们把这家公司及其提供的软件都简称为**MapR**。读者可通过http://www.mapr.com下载MapR，它虽然同样以Hadoop为基础，但进行了一些改进。

MapR主要关注软件性能和可用性。我们曾在第7章讲过，Hadoop NameNode和JobTracker是Hadoop核心组件的薄弱环节，而MapR率先实现了Hadoop NameNode和JobTracker的高可用性解决方案。MapR还集成了NFS文件系统，这样就可以更简便地处理现有数据。MapR用完全兼容POSIX的文件系统代替了HDFS，便于远程挂接。

MapR同时提供了免费版和企业版软件，但免费产品只能使用其部分扩展功能。一旦用户购买了企业版软件，公司会提供相应的技术支持、培训和咨询服务。

- IBM InfoSphere Big Insights

本节讲的最后一个产品来自IBM公司。读者可通过http://www-01.ibm.com/software/data/infosphere/biginsights/下载到**IBM InfoSphere Big Insights**的安装包。和MapR一样，它也对开源Hadoop的核心组件进行了商业改进和扩展。

IBM分别提供了Big Insights的免费版和企业版。免费版对Apache Hadoop产品进行了改进，新增了一些免费的管理工具和部署工具，同时还集成了一些其他IBM软件。

企业版与免费版的差别较大，它不仅仅建立在Hadoop基础之上，实际上，它还可以与CDH

或HDP共同使用。企业版提供了一系列的数据可视化、商业分析和处理工具。它还深度集成了IBM的其他产品，如InfoSphere Streams、DB2和GPFS。

3. 如何选择合适的产品

可以看出，用户的选择范围很大，从方便安装的完全开源的集成产品，到全部定制的集成分析工具。每种产品都各有千秋，用户在选择使用哪款产品时要仔细考虑具体需求。因为上述产品都提供了基本版本的免费下载，因此用户可以先试用一下，然后再选择合适的产品。

11.4　其他 Apache 项目

无论用户用的是集成产品，还是基本的Apache Hadoop，都会经常遇到需要使用其他相关的Apache项目的情况。本书已经介绍了Hive、Sqoop和Flume，接下来将重点介绍其他项目。

需要注意的是，本节内容重在突出一些重点项目（根据我的理解），同时尽量介绍更多的各类可用项目。读者的思路要开阔一些，任何时候都有新项目发布。

11.4.1　HBase

HBase也许是最受欢迎但本书尚未提到的与Hadoop相关的Apache项目，其主页地址为http://hbase.apache.org。它是一款以HDFS为基础的非关系数据库，其理论基础是Google在一篇学术论文中提到的BigTable数据存储模型。

鉴于MapReduce和Hive任务都侧重于成批地访问数据，HBase正好相反，它力求低延迟的访问数据。因此，与之前提到的其他技术不同，HBase能够直接支持面向用户的服务。

HBase并没有像Hive和其他RDBMS那样采用关系型模型，而是采用了面向列的非结构化键值模型。用户可以在HBase运行时新增数据列，并依据数据值将其插入HBase中的合适位置。因为HBase实现了从原始键到目标列的高效映射，所以每次数据查找操作的执行速度都很快。HBase把时间戳当做数据的另一个属性，因些可以直接根据时间点检索数据。

这种数据模型的功能非常强大，但并不适用于所有情况，就像关系模型也无法做到普遍适用一样。但如果读者需要低延迟地访问存储在Hadoop里的大规模结构化数据，HBase绝对是最好的选择之一。

11.4.2　Oozie

我们曾多次讲过，Hadoop集群并非存在于真空中，而要与其他系统集成以扩展工作流的概念。Oozie是一个针对Hadoop开发的工作流调度工具，其主页地址为http://oozie.apache.org。

Oozie以最简单的模式提供了执行MapReduce作业的调度机制，其调度原则要么是一定的时间段（例如，每小时执行一次），要么是数据可用性（例如，当新数据到达时执行）。Oozie还可以指定多级工作流，它们可以描述一个完整的端到端流程。

除了直接调度MapReduce作业的运行之外，Oozie也可以调度Hive或Pig命令的运行，甚至是完全和Hadoop无关的任务（如发邮件、执行shell脚本或在远程机上运行命令）。

用户可通过多种方式构建工作流，一种通用的方法就是使用**Pentaho Kettle** (http://kettle.pentaho.com)和**Spring Batch**（http://static.springsource.org/spring-batch）这样的**Extract Transform Load**（ETL）工具。这些工具中集成了部分Hadoop，但传统的专用的工作流引擎可能并不包含这些集成。如果读者要构建一个Hadoop交互工作流，手边却没有一款满意的集成了Hadoop的工作流工具，不妨考虑一下Oozie。

11.4.3　Whir

如果读者想借助Amazon AWS这样的云服务部署Hadoop，还不如直接使用更高级的ElasticMapReduce服务，而不必再基于EC2搭建自有集群。尽管Amazon提供了帮助脚本，但在云基础设施上部署Hadoop依然非常复杂。这就是开发Whir的初衷，其主页地址为http://whir.apache.org。

Whir并非专用于解决Hadoop的部署问题，它是一个通用的云服务，并不限定具体的应用程序，Hadoop只是其中一个例子。Whir提供了一种程序化的方法在云基础设施上部署Hadoop，它会解决所有底层服务的问题。而且，它与云服务提供商无关，因此，假如读者在Amazon提供的EC2主机上完成了Hadoop的配置，那就可以用相同代码在其他云上（如Rackspace或Eucalyptus）完成同样的设置。这样就会避免在不同云服务平台上部署应用程序的很多问题。

Whir做的还远远不够。现如今它支持的服务种类有限，并且只支持Amazon提供的AWS。然而，如果读者想更顺利地完成云端部署，有必要了解一下Whir。

11.4.4　Mahout

上述项目都是通用的，它们支持任何领域的应用程序。而Apache Mahout则是基于Hadoop和MapReduce创建的机器学习算法库，其主页地址为http://mahout.apache.org。

Hadoop的数据处理模型非常适合于机器学习应用程序，其目标即是从大规模数据集凝练数据价值和数据意义。Mahout实现了很多常用的机器学习技术，例如聚类和推荐系统。

如果读者要从大量数据中找出关键的数据模式、数据关系，或仅仅是像大海捞针一样找出某个目标值，Mahout也许能够发挥作用。

11.4.5　MRUnit

我们即将介绍的最后一个Apache Hadoop项目再次突出了围绕Hadoop开发的应用程序种类之多。在很大程度上，如果MapReduce作业经常由于隐藏的bug而失败，那么读者用到了多少新技术或用了哪个版本的安装包已变得毫无意义。最近推出的MRUnit可帮助解决这个问题，其主页地址为http://mrunit.apache.org。

开发MapReduce作业有一定的困难，尤其是在早期阶段，但对它们进行测试和调试一直都令人非常头疼。顾名思义，MRUnit采用了JUnit和DBUnit这两种单元测试模型，并提供了一个有助于编写和执行测试代码的框架，可帮助用户提高代码质量。构建测试集、自动化测试集成工具、编译工具以及所有软件工程的最好做法,MRUnit提供了读者编写非MapReduce代码时做梦都想拥有的这些功能。

如果读者曾编写过任何MapReduce作业，就会觉得MRUnit很有意思。依我个人浅见，它是一个非常重要的项目，值得读者深入研究。

11.5　其他程序设计模式

人们不仅开发了扩展Hadoop功能的应用程序，同时也出现了其他编程工具，它们遵循全新的编程范式。

11.5.1　Pig

我们在第8章提到过Pig（http://pig.apache.org），所以本节不再多讲其他内容。一定要记住，与编写原始MapReduce代码或HiveQL脚本相比，定义Hadoop处理过程的数据流更为直观且更适合。Pig与Hive的主要区别是：Pig是一个命令式语言，它定义了数据处理过程的执行方式，而Hive更侧重于描述，它定义了目标结果却不管如何实现这个目标。

11.5.2　Cascading

Cascading并不是一个Apache项目，但它也是开源的，其主页地址为http://www.cascading.org。Hive和Pig使用不同语言描述数据处理过程，而Cascading则提供了一系列更高层的抽象。

它无需考虑多个MapReduce作业的处理方式以及如何与Cascading共享数据，而是采用了数据流的数据模型。数据流中用到了管道和多个连接器、分接头和类似构件。这些构件是以编程方式构建的（原始的核心API使用Java语言，但也用许多其他语言重写了这些API），Cascading负责管理工作流在集群上的翻译、调度和执行。

如果读者想使用更高级的MapReduce接口，而Pig和Hive的陈述性风格又不太合适，Cascading的编程模型可能正是读者想要的。

11.6　AWS 资源

许多Hadoop技术可以部署在AWS上，作为自管理集群的一部分功能。Amazon提供了对弹性MapReduce的支持，它把Hadoop视为一个托管服务。与此类似，还有一些其他服务也可以实现这种托管。

11.6.1　在EMR上使用HBase

事实上，这并不是一个独特的服务，但是就像EMR本身就支持Hive和Pig服务一样，它也可以对HBase集群提供直接支持。这是一个相对较新的功能，还需要观察一下它在实际工作中的效果如何。但根据以往经验，HBase对网络质量和系统负载较为敏感。

11.6.2　SimpleDB

Amazon推出的SimpleDB服务（http://aws.amazon.com/simpledb）提供了类似HBase的数据模型。事实上，它并非建立在Hadoop基础之上，但它和dynamodb服务都提供了托管服务，如果读者对类似HBase的数据模型感兴趣的话，可以考虑使用它们。该服务早在几年前就已经推出，实践表明，它对使用案例的理解已经非常成熟。

但是，SimpleDB也存在一些缺陷，尤其是对表的大小有限制，并且需要手工对大数据集进行分区。但如果读者只需使用HBase类型系统存储小规模数据，这将是一个不错的选择。并且该系统易于配置，非常适合用于处理基于列的数据模型。

11.6.3　DynamoDB

近期，AWS推出了一种名为**DynamoDB**的新服务，其主页地址为http://aws.amazon.com/dynamodb。虽然DynamoDB和SimpleDB、HBase的数据模型非常相似，但它针对的是其他应用类型。另一个区别是，SimpleDB提供了非常丰富的查询API，但受限于数据大小，而DynamoDB提供的API数量有限却几乎能够支持任意规模的数据。

DynamoDB的定价模式很有意思，用户不是按照使用的服务器数量支付费用，而是由用户申请的读写容量以及DynamoDB为满足这个需求而提供的资源来计费。这种纯粹的服务模型很有意思，用户完全不了解系统是如何实现预期性能的。如果读者要存储的数据规模远远超出了SimpleDB的能力范围，不妨考虑下DynamoDB，但也要好好考虑其定价模型，因为申请太多的空间会导致用户成本急剧上升。

11.7 获取信息的渠道

无论这些新技术和新工具功能如何强大,用户可不光只需要这些新技术和新工具。某些时侯,来自有经验人员的小小帮助会解决用户的困惑。读者不必担心这方面的问题,Hadoop开发团队在很多领域的能力都很强。

11.7.1 源代码

读者有时容易忽视这个现象:Hadoop和其他Apache项目都是完全开源的。这些项目的源代码从根本上解释了系统的工作原理。熟悉源代码并且跟踪某些功能的实现方式可以为用户提供巨大的信息量,这在用户遇到意外情况的时候能够提供巨大的帮助。

11.7.2 邮件列表和论坛

几乎前几节列出的所有项目和服务都建立了自己的邮件列表和(或)论坛,读者可在相应项目主页查看特定链接。如果读者使用的是AWS,请通过https://forums.aws.amazon.com访问AWS开发者论坛。

勿必要仔细阅读相关的用户指南,并了解预期的行为。这些指南都能够提供大量信息,包括开发者经常访问的某些项目的邮件列表和论坛。用户可以在Hadoop邮件列表找到Hadoop核心组件的开发人员,在Hive邮件列表找到Hive开发人员,在EMR论坛找到EMR开发人员,等等。

11.7.3 LinkedIn群组

在LinkedIn社交网络上存在很多Hadoop和相关群组。读者可以搜索自己感兴趣的特定领域,但更好的办法是访问通用Hadoop用户组,其网址为http://www.linkedin.com/groups/Hadoop-Users-988957。

11.7.4 Hadoop用户群

如果读者想获得更多面对面的交流,可以参加本领域的**HUG**(Hadoop User Group,Hadoop用户群),大部分这些用户群都列在了http://wiki.apache.org/hadoop/HadoopUserGroups网页上。这些Hadoop用户群通常会安排一些非正规聚会,可以一边听到高质量的演讲,一边手拿披萨和饮料与志趣相投的人讨论技术。

如果读者住的地方附近没有HUG的话,可以考虑组织一个!

11.7.5　会议

虽然Hadoop是一门相对较新的技术，但该领域已经举办了一些重要的会议活动，会议主题包括开源项目、学术研究和商业应用。**Hadoop Summit**会议的影响力非常大，关于它和另外会议的介绍参见http://wiki.apache.org/hadoop/Conferences。

11.8　小结

本章快速介绍了更广泛的Hadoop生态系统。我们依次介绍了Hadoop即将发生的变化，尤其是HDFS高可用性和YARN，以及存在其他版本Hadoop安装包的原因，并列举了一些较受欢迎的版本，还有其他扩展Hadoop功能或为其提供支持的Apache项目。

我们还介绍了编写或创建Hadoop作业的其他方法，如何获取相关信息，以及怎样与其他Hadoop爱好者取得联系。

现在尽情享受使用Hadoop的乐趣，并创建出让人惊叹的杰作吧！

随堂测验答案

第 3 章　理解Map Reduce

随堂测验：键值对

Q1	2
Q2	3

随堂测验：MapReduce的结构

Q1	1
Q2	3
Q3	(2) 不能使用Reducer C作为combiner。如果使用Reducer C作为combiner，combiner会向最终的reducer输出一系列平均值，而reducer却不知道这些平均值是通过多少个项目的数据计算出来的，意味着它无法计算整体平均值。单纯从选取每个键的最大值和最小值的角度来讲，Reducer D是可以用于combiner操作的。但如果想要计算每个键的最大值和最小值之间的方差，这就行不通了。如果某个键的最大值周边数据分布较为集中的话，方差就比较小。类似地，如果最小值周边数据分布较为集中的话，方差也比较小。这些子区间内存在少数几个孤立值，因此最终reducer还是无法算出想要的结果

第 7 章　系统运行与维护

随堂测验：集群配置

Q1	(5) 尽管可以依据通用规则决定集群的硬件配置，但最好想想集群会用于处理哪种作业，最佳配置最终还是取决于数据处理任务
Q2	(4) 网络存储较受欢迎，但很多情况下，读者会发现由数百台主机组成的大型Hadoop集群却依赖于单一存储设备。这就为集群带来了新的失败风险，而且这一风险比其他风险的可能性更大。人们通常使用硬盘冗余技术解决存储失效问题。这些硬盘阵列可以达到很高的性能，但读写成本较大。让Hadoop自行解决失效问题，赋予其在同样数量的硬盘上进行并行存取的能力，可以达到更高的性能

（续）

Q3	(3) 我们建议尽量不要采用第一种配置。尽管这种配置方案提供了足够的存储空间，但其数据处理能力不足，为了降低硬件升级的频率，我们可以选择更优的方案。随着数据量的增长，马上就需要新增主机处理这些数据。而MapReduce作业复杂性的提升，又对处理器能力和内存提出了更高的要求 配置方案B和C都不错，因为它们提供了过剩的存储空间可应对数据量的增长，并提供了相似的处理器和内存上限。方案B的硬盘I/O速率较高，而方案C的CPU性能更好。因为主作业涉及金融建模和预测，我们假设每个任务都对CPU和内存提出了严格的要求。配置方案B的I/O性能稍高一些，但假如处理器利用率达到了100%，那么另一个硬盘就得不到有效利用。因此，处理器性能更好的主机似乎更适合用在本例中 配置方案D远超出了任务需求，这也正是没有选用该方案的原因。我们为何要多花钱购买远超过实际需要的硬件呢

亚马逊读者评论

“本书能够帮助读者更好地理解Hadoop，并学会如何使用Hadoop。”

“本书讲解全面，解释清晰，对初学者来说是一个非常不错的选择。”

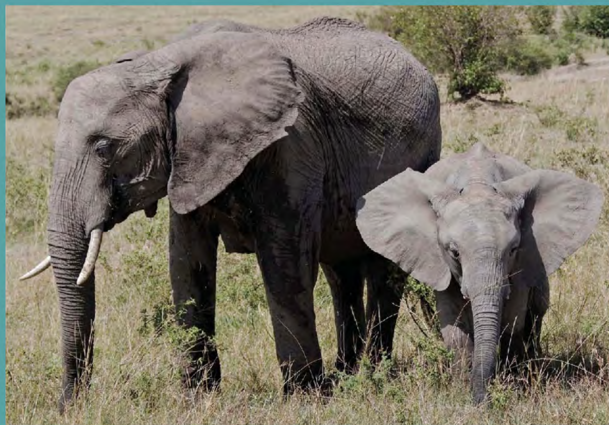
本书揭开了Hadoop的神秘面纱，它着重讲解了如何应用Hadoop和相关技术搭建工作系统并完成任务。在读者明白这些内容之后，又介绍了如何使用云服务完成相同任务。本书从Hadoop的基本概念和初始设置入手，讲述了如何开发Hadoop程序，如何在数据规模增长的时候维持系统运行，本书涵盖了有效使用Hadoop处理实际问题所需用到的全部知识。

主要内容：

- Hadoop和云服务出现的历史背景，以及何时适用Hadoop的背景知识；
- 安装并配置Hadoop集群的最佳方式，根据手头的问题调整系统配置；
- 用Java和Ruby示例程序讲解如何编写运行在Hadoop上的程序；
- Amazon网络服务提供的托管Hadoop集群的运行方式，以及它与用户直接管理的Hadoop集群有何区别；
- Hadoop与关系数据库的融合，使用Hive执行SQL查询，使用Sqoop迁移数据；
- 组成Hadoop生态系统的其他项目和工具，以及Hadoop的发展方向。

针对初学者的有效方法

- 通过清晰操作步骤讲解最有用的任务
- 边干边学——立刻动手实践
- 摆脱枯燥的二进制
- 有启发意义的理想的案例，能够带给读者灵感，从而解决面临的问题
- 促进读者动手练习的作业和习题



[PACKT] PUBLISHING

图灵社区：iTuring.cn
热线：(010)51095186 转 600
分类建议 计算机/程序设计
人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-34133-4
定价:65.00元